**Exceptions**

**Generics**

**Collection Framework**

**RegEx**

Excpetion:

AN unforeseen event that occurs during the execution of program

It is run time error that terminates the execution of program

ExceptionHandling.java

```java
package javalangfeatures;

public class ExceptionHandling {

    public static void main(String[] args) {

        int[] arr = { 1, 2, 3 };
        int index = 12;
        System.out.println(arr[index]);

        /*
         * Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 12
         * out of bounds for length 3 at
         * javalangfeatures.ExceptionHandling.main(ExceptionHandling.java:9)
         */


        int a = 0;
        int c = 10 / a;// a run time error occurs
        System.out.println(c);// not exceuted
        /*
         * Exception in thread "main" java.lang.ArithmeticException: / by zero at
         * javalangfeatures.ExceptionHandling.main(ExceptionHandling.java:7)
         */

        //these above examples leads to abruptly ends the program execution
        //to overcome this we have exception handling
    }

}
```

```java
package javalangfeatures;

public class ExceptionHandling1 {

    public static void main(String[] args) {
        // when run time error occurs in a program
        // programmer can define the way to handle exception
        // if not JVM will handle exception and throw some exception , which is not
        // easily understood

        // it is good practice to handle exception
        // to handle exceptions we have try catch finally throw throws

        // try block : stmts which are excepted to throw error is placed in catch block
        // catch block: follows try block and appropriate exception catch will executes
        // , if no exception then catch will not execute

        int a = 10;
        int b = 0;
        try {
            int c = a / b;
        } catch (ArithmeticException e) {
            System.out.println("Please donot use denominator as zero");
        }

        System.out.println(a);
        System.out.println(b);

    }

}
```

Problems Javadoc Declaration Console × Debug TestNG

```
<terminated> ExceptionHandling1 [Java Application] D:\Program Files\Eclipse\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.
Please donot use denominator as zero
10
0
```

```java
package javalangfeatures;

public class ExceptionHandling1 {

    public static void main(String[] args) {
        // when run time error occurs in a program
        // programmer can define the way to handle exception
        // if not JVM will handle exception and throw some exception , which is not
        // easily understood

        // it is good practice to handle exception
        // to handle exceptions we have try catch finally throw throws

        // try block : stmts which are excepted to throw error is placed in catch block
        // catch block: follows try block and appropriate exception catch will executes
        // , if no exception then catch will not execute
//no two catch blocks can handle same exception , Exception should be present last
        int a = 10;
        int b = 0;
        try {
            int c = a / b;
        }catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
            System.out.println("either divided by zero or array index out of bounds");
        }
        catch(Exception e) {
            System.out.println("SOme unknown error happened");
        }

        System.out.println(a);
        System.out.println(b);

    }

}
```

```java
package javalangfeatures;

public class ExceptionHandling3 {

    public static void main(String[] args) {
        // finally block executes always
        // we can use try and finally
        try {

        }catch(Exception e){

        }finally {

        }


        try {

        }finally {

        }

    }

}
```

```
/*
 * Throwable is parent class of all exceptions
 * This has 2 child Exception and Error
 *
 * Exception has RuntimeException which inturn has ArrayIndexOutOfBounds , Arthimetic , NullPointer ,...
 *
 *
 * Throwable class is super class of all exceptions in java
 * Throwable has following methods
 */
```

# Throwable and Exception class

- The **Throwable** class is the parent class of all the exceptions in Java
- It has following important methods:

| Method | Description |
|---|---|
| String getMessage() | Returns a description of the exception |
| void printStackTrace() | Displays the stack trace |
| String toString() | Returns a String object containing a description of the exception |

- The **Exception** class is subclass of Throwable class but it does not define any methods of its own

```
    int totalSalaryPerDepartment[] = {120000, 240000, 360000, 280000, 190000, 2100000, 370000,150000 };

    int numberOfEmployeesPerDepartment[] ={5, 6, 7, 9, 7, 0, 8, 5 };

    try
    {
        for(int i=0; i<=totalSalaryPerDepartment.length;i++)
        {
            int averageSalary = totalSalaryPerDepartment[i]/numberOfEmployeesPerDepartment[i];
            System.out.println("Average Salary is: "+averageSalary);
        }

    }
    catch (ArithmeticException exception)
    {
        System.out.println("\nPlease don't use zero as denominator ");
        System.out.println(exception.getMessage());
    }
    catch (ArrayIndexOutOfBoundsException exception)
    {
        System.out.println("\nPlease don't access array beyond the last index");
        System.out.println(exception.getMessage());
    }
    catch (Exception exception)
    {
        System.out.println("\nSome unknown error happened");
        System.out.println(exception.getMessage());
    }
}
```

```java
int totalSalaryPerDepartment[] = {120000, 240000, 360000, 280000, 190000, 2100000, 370000,150000 };

int numberOfEmployeesPerDepartment[] ={5, 6, 7, 9, 7, 0, 8,5 };

try
{
    for(int i=0; i<=totalSalaryPerDepartment.length;i++)
    {
        int averageSalary = totalSalaryPerDepartment[i]/numberOfEmployeesPerDepartment[i];
        System.out.println("Average Salary is: "+averageSalary);
    }

}
catch (ArithmeticException exception)
{
    System.out.println("\nPlease don't use zero as denominator ");
    exception.printStackTrace();
}
catch (ArrayIndexOutOfBoundsException exception)
{
    System.out.println("\nPlease don't access array beyond the last index");
    exception.printStackTrace();
}
catch (Exception exception)
{
    System.out.println("\nSome unknown error happened");
    exception.printStackTrace();
}
}
```

printStacktrace useful for debugging

# User-defined Exception

- The user can create his own exception classes by extending Exception class
- User-defined exceptions are customized exceptions based on the user's need
- For example, the following code creates InsufficientBalanceException:

```java
public class InsufficentBalanceException extends Exception {
    public InsufficentBalanceException(){
        super("Insufficient Balance");
    }
}
```

## throw statement

- The programmer can throw an exception using the **throw** statement
- Only an object of the Throwable class or its sub classes can be thrown
- Program execution abruptly stops on encountering throw statement, and the catch statement or statements are checked for the matching type of exception
- **Syntax:**

  **throw** throwableObject;

  – Here, *throwableObject* is an object of Exception class or its subclasses

## throws statement

- This statement is used to inform the programmer that a particular method can throw an exception
- Any method which can cause exceptions must list all the exceptions possible during its execution for the programmer's knowledge using the throws keyword
- This statement makes sure that the calling method handles the exception being thrown by the current method
- **Syntax:**

  Return-type methodName (argument-list) **throws** exceptionName1, exceptionName2,.....

```
{
   //method code
}
```

ExceptionDemo1.java | InsufficientBalanceException.java

```java
package demo2;
class Account {
    private double balance;
    private String accountNumber;

    public Account(double balance, String accountNumber) {
        this.balance = balance;
        this.accountNumber = accountNumber;
    }
    public double withdraw(double amount) throws InsufficientBalanceException {
        if (amount > balance) {
            throw new InsufficientBalanceException();
        } else {
            balance = balance - amount;
        }
        return balance;
    }
}
```

ExceptionDemo1.java | InsufficientBalanceException.java

```java
package demo2;

public class InsufficientBalanceException extends Exception{

    public InsufficientBalanceException(){

        // Exception message
        super("Insufficient Balance");

    }

}
```

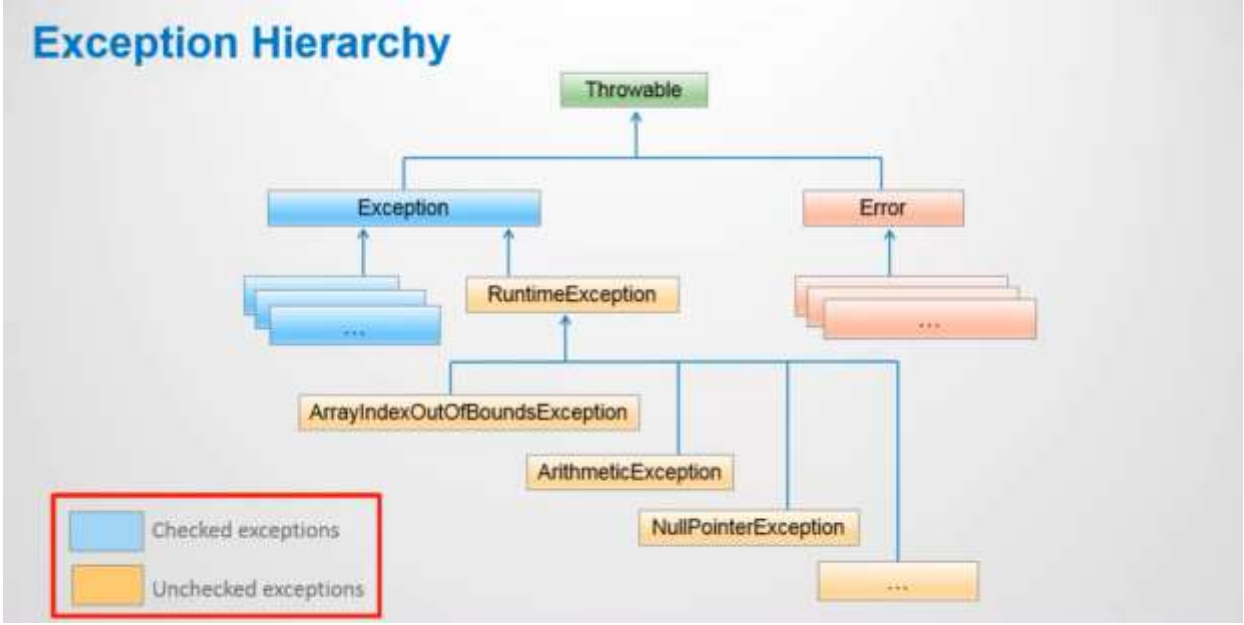
```java
    private String accountNumber;

    public Account(double balance, String accountNumber) {
        this.balance = balance;
        this.accountNumber = accountNumber;
    }
    public double withdraw(double amount) throws InsufficientBalanceException
        if (amount > balance) {
            throw new InsufficientBalanceException();
        } else {
            balance = balance - amount;
        }
        return balance;
    }
}

public class ExceptionDemo1 {

    public static void main(String args[]) throws InsufficientBalanceException {

        Account account = new Account(2000, "1234567890");

        account.withdraw(2100);

        System.out.println("Executed Successfully");
    }
}
```

Exception Propagation

Runtime

```java
public class ExceptionDemo1 {

    public static void main(String args[]) {

        Account account = new Account(2000, "1234567890");
        try {

            account.withdraw(2100);

        } catch (InsufficientBalanceException e) {
            System.out.println("Exception Message: "+e.getMessage());
        }
        System.out.println("Executed Successfully");
    }
```

```java
    public Account(double balance, String accountNumber) {
        this.balance = balance;
        this.accountNumber = accountNumber;
    }
    public double withdraw(double amount) throws InsufficientBalanceException,Exception {
        if (amount > balance) {
            throw new InsufficientBalanceException();
        } else {
            balance = balance - amount;
        }
        return balance;
    }
}

public class ExceptionDemo1 {

    public static void main(String args[]) {

        Account account = new Account(2000, "1234567890");
        try {

            account.withdraw(2100);

        } catch (InsufficientBalanceException e) {
            System.out.println("Exception Message: "+e.getMessage());
        } catch (Exception e) {
            System.out.println("Some unknown Exception happened");
        }
        System.out.println("Executed Successfully");
    }
```



## Checked Exceptions

- Exception class and all classes directly inherited from it (except the RuntimeException class) are **checked exceptions**
- Compiler forces the programmer to handle or declare them
- Some common checked exceptions are:

| Exception | Description |
|---|---|
| ClassNotFoundException | No definition for class is found. |
| IOException | I/O operation is interrupted or failed. |

## Unchecked Exceptions

- **RuntimeException** class and all classes inherited from it are **unchecked exceptions**
- Compiler does not force the programmer to handle them
- Some common unchecked exceptions are

| Exception | Description |
|---|---|
| ArithmeticException | Arithmetic error, such as division by zero |
| ArrayIndexOutOfBoundsException | Array index is negative or greater than array size |
| NullPointerException | Use of null reference instead of object |
| ClassCastException | Cast is invalid |

Generics:

## Introduction

- Consider the following list: (employeeNameList) used to store EmployeeNames:
  - **List** employeeNameList **= new ArrayList ()**

```
for(int i=0; i<list1.size();i++){
    int len= ((String)list1.get(i)).length();
    System.out.println(len);
}
```

**RUN TIME ERROR**

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
    at demo1.ExceptionDemo.main(ExceptionDemo.java:10)
```

| employeeNameList |
|---|
| Jack |
| Peter |
| 12 |
| Justin |
| 14 |
| 15 |

**HOW TO RESTRICT A LIST TO STORE ONLY ONE TYPE OF DATA ???**

## Generics in Java

- Consider the following list: (employeeNameList) used to store EmployeeNames
  - **List<String>** employeeNameList **= new ArrayList<String>()**

Type Safety using Generics

Can contain Strings only

**employeeNameList**

## Generics in Java

- Consider the following list: (employeeNameList) used to store EmployeeNames
  - **List<String>** employeeNameList **= new ArrayList<String>()**
  - employeeNameList.add("Jack");
  - employeeNameList.add("Eric");
  - employeeNameList.add(1002);

**COMPILE TIME ERROR**

Can contain Strings only

**employeeNameList**

Jack

Eric

**GENERICS HELP IN CONSTRAINING THE DATA TYPES**

```java
package javalangfeatures;

import java.util.ArrayList;
import java.util.List;

public class GenericsDemo {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<String> EmplyeeNames = new ArrayList<String>();

        EmplyeeNames.add("Apple");
        EmplyeeNames.add("Mango");
        EmplyeeNames.add(1);// this throws compile time error The method add(int, String) in the type
                            // List<String> is not applicable for the arguments (int)
    }

}
```

```java
package demo;

import java.util.ArrayList;
import java.util.List;

public class Demo1 {
    public static void main(String[] args) {
        List employeeNameList  = new ArrayList();

        employeeNameList.add("Jack");
        employeeNameList.add("Eric");
        employeeNameList.add(12);
        employeeNameList.add("Dan");

        System.out.println("List is: "+employeeNameList);
```

Console

```
<terminated> Demo1 (3) [Java Application] Q:\JAVALABS\001\Java\jre6\bin\javaw.exe (Jun 10, 2015 11:18:25 AM)
List is: [Jack, Eric, 12, Dan]
```

```java
package demo;

import java.util.ArrayList;
import java.util.List;

public class Demo1 {
    public static void main(String[] args) {
        List employeeNameList  = new ArrayList();

        employeeNameList.add("Jack");
        employeeNameList.add("Eric");
        employeeNameList.add(12);
        employeeNameList.add("Dan");

        System.out.println("List is: "+employeeNameList);

        System.out.println("\n\nLength of all the elements:");
        System.out.println("==========================");
        for (int i = 0; i < employeeNameList.size(); i++) {
            int len = ((String)employeeNameList.get(i)).length();
            System.out.println(len);

        }

    }

}
```

Console

```
<terminated> Demo1 (3) [Java Application] Q:\JAVALABS\001\Java\jre6\bin\javaw.exe (Jun 10, 2015 11:25:27 AM)
Exception in thread "main" List is: [Jack, Eric, 12, Dan]


Length of all the elements:
==========================
4
4
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
        at demo.Demo1.main(Demo1.java:20)
```

```java
package demo;

import java.util.ArrayList;

public class Demo2 {
    public static void main(String[] args) {

        List<String> employeeNameList = new ArrayList<String>();

        employeeNameList.add("Jack");
        employeeNameList.add("Eric");
        employeeNameList.add("Dan");



        System.out.println("List is: " + employeeNameList);

        System.out.println("\n\nLength of all the elements:");
        System.out.println("----------------------------");
        for (int i = 0; i < employeeNameList.size(); i++) {
            int len = ((String) employeeNameList.get(i)).length();
            System.out.println(len);

        }

    }

}
```

Generics

Generics provides type safety and any conflicts that can occur due to datatypes

## Generics with Classes

- Generics can be used while creating classes and interfaces

Example:

```
class MyList<T>{
  private T[] elementList;
  //Get and Set methods are coded
}
```

Here "T" is representing any **Type Parameter Name.**

Actual value of T can be any Data type Like: Integer, Float etc. provided at the time when object is created

- Examples of valid usage
  - MyList list = new MyList();
  - MyList<Integer> list1 = new MyList<Integer>();
  - MyList<Double> list2 = new MyList<Double>();

Raw type

Value of Type Parameter T is Integer so this object can deal with only Integer Values

Value of Type Parameter T is Double so this object can deal with only Double Values

Generics are allowed with reference types but not with implicit types like: int, float etc.

- Examples of Invalid usage
  - MyList<Integer> integerList1 = new MyList<String>();
  - MyList<Number> integerList2= new MyList<Integer>();

Data type is not same

Number class is parent of Integer class in Java

Data type is not same and Generics don't support parent type pointing to the child type

Generic Class:

```
package demo;

class MyList<T>{
    private T[] elementList;

    public T[] getElementList() {
        return elementList;
    }

    public void setElementList(T[] elementList) {
        this.elementList = elementList;
    }

}
public class Demo3 {
    public static void main(String[] args) {
        MyList<Integer> list = new MyList<Integer>();

        Integer[] integerArray= {1,2,3,4};
        Double[] doubleArray= {1.1,2.1,3.1,4.1};

        list.setElementList(integerArray);
        list.setElementList(doubleArray);

    }

}
```

```java
package javalangfeatures;


class MyList<T>{
    private T[] elementsList;

    public T[] getElementsList() {
        return elementsList;
    }

    public void setElementsList(T[] eleList) {
        this.elementsList = eleList;
    }
}
public class GenericsWithClasses {

    public static void main(String[] args) {
        MyList<Integer> obj = new MyList<Integer>();
        Integer[] nums = {1,2,3};
        obj.setElementsList(nums);
        System.out.println(obj.getElementsList());

    }

}
```

## Generics with Methods

- **Syntax:**

  **<type-parameters>** return-type method-name(parameter-list){

  // method body

  }

- Generic methods are invoked by prefixing method name with actual type in angle brackets
  - Example: object.<type-parameter>methodName(parameter-list);

```java
package demo;


class PrintDetails {
    public <T> void displayDetails(T[] array) {
        for (T o : array) {
            System.out.print(o+"\t");
        }
    }
}

public class Demo4 {
    public static void main(String[] args) {

        String[] employeeNameList = {"Jack","Eric"};
        Integer[] employeeIdList = {1001,1002,1003,1004};

        PrintDetails details = new PrintDetails();

        System.out.println("Displaying the EmployeeId List");
        System.out.println("--------------------------------");
        details.displayDetails(employeeNameList);

        System.out.println("\n\nDisplaying the EmployeeName List");
        System.out.println("--------------------------------");
        details.displayDetails(employeeIdList);

    }
```

equivalent to details.<String>displayDetails(employeeNameList);

equivalent to details.<Integer>displayDetails(employeeIdList);

Generic Class Demo:

```java
class Record<E> {
    private E record;

    public void display(E item) {
        System.out.println(item);
    }
}

class Student {
    private int studentId;
    private String studentName;

    public Student(int studentId, String studentName) {
        this.studentId = studentId;
        this.studentName = studentName;
    }

    public String toString() {
        return "Student: Id = " + studentId + " Name = " + studentName;
    }
}

class GenericsDemo {
    public static void main(String[] args) {
        Student s1 = new Student(101, "Robert");
        Record<Integer> integerRecord = new Record<Integer>(); // integerRecord can be used to display only integers
        integerRecord.display(12);
        // integerRecord.display(s1); will give an error as we are trying to add a
        // student class object
        Record<Student> studentRecord = new Record<>(); // studentRecord can be used to display only Students
        studentRecord.display(s1);
        // studentRecord.display(15); will give an error as we are trying to add an
        // integer
    }
}
```

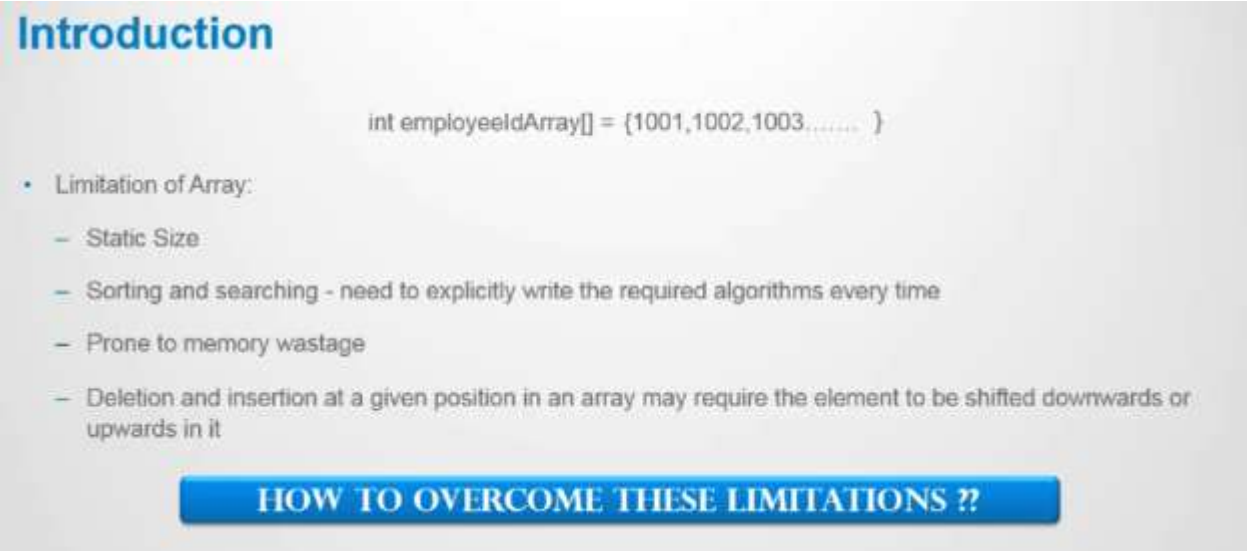
Generic Methods:

```java
class UserInterface {
    public static <E> void display(E[] list) {    // generic method displaying the elements of an array
        for (int i = 0; i < list.length; i++)
            System.out.println(list[i] + ", ");
    }
    public static void main(String[] args) {
        String[] cities = {"Bengaluru", "Chennai"};
        Integer[] codes = {12,14,15};
        UserInterface.display(codes);
        UserInterface.display(cities);

    }

}
```

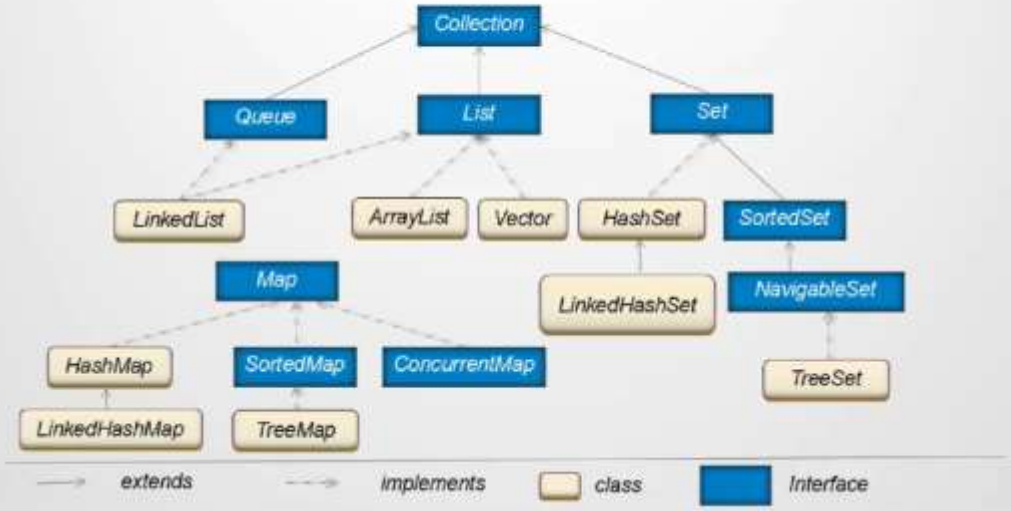
Collection Framework in Java:

In a company we have so many employees have to store all empIds , array size is fixed , addition deletion is time consuming in array

This can be overcome by Collections



# Collection Framework




# Collection Framework – Interfaces

- Collection – the root of collection hierarchy, represents a collection of objects called elements
- List – an ordered collection, can have duplicate values , can control where in the list an element is positioned, an element can be accessed based on their position (index)
- Set – a collection that cannot contain duplicate elements
- **Map** – an object that maps keys to values, cannot have duplicate keys; each key can map to at most one value

| List |
|---|
| 11 |
| 11 |
| 33 |
| 14 |
| 12 |
| 14 |

| Set |
|---|
| 11 |
| 22 |
| 33 |
| 14 |
| 12 |

| Map | |
|---|---|
| Key | Value |
| 1 | "Jack" |
| 2 | "Jill" |
| 3 | "Jerry" |

## Collection Framework : Common Methods

| Method | Description |
|---|---|
| add(element):boolean | Adds the element to the collection |
| addAll(Collection c):boolean | Adds all the elements of the received collection to the current collection |
| remove(element):boolean | Removes the received element from the current collection |
| removeAll(Collection c):boolean | Removes all the elements of the received collection from the current collection |
| contains(element):boolean | Returns true if the collection contains the element |
| retainAll(Collection):boolean | Retains all the elements of the received collection in the current collection and deletes the rest of the elements in the current collection |

## Collection Framework : Common Methods (contd..)

- add(), addAll(), remove(), removeAll()

| | list1 |
|---|---|
| list1.add(10); | 10 |
| list1.add(5); | 5 |
| list1.add(20); | 20 |
| list1.addAll(list2); | 15 |
| | 9 |

| list2 |
|---|
| 15 |
| 9 |

If present and removed returns true and viceversa

false ← returns list1.remove(21);

| list1 |
| --- |
| 10 |
| 5 |
| 15 |
| 9 |

list1.removeAll(list2);

| list1 |
| --- |
| 10 |
| 5 |
| 15 |
| 9 |

| list2 |
| --- |
| 5 |
| 15 |

true ← returns list1.contains(10);

| list1 |
| --- |
| 10 |
| 9 |
| 12 |
| 14 |
| 34 |


## List interface

- It extends Collection interface
- It allows duplicate elements which can be accessed based on their index position
- Methods specific to List:

| Name | Description |
|---|---|
| void add(int index, Object o) | Adds an object at specified index |
| boolean addAll(int index, Collection c) | Adds all the objects of the received collection to the current list at specified index |
| Object get(int index) | Returns the object at specified index |
| Object remove(int index) | Removes the object at the specified index |
| ListIterator listIterator() | Returns a list iterator over the objects in the list |

- Implementation classes
  - ArrayList, LinkedList

# ArrayList

- Array implementation of List interface
- Allows random access as it works on the basis of index
- Adding and removing elements is time consuming

```java
package demo1;

import java.util.ArrayList;

public class Demo1 {
    public static void main(String[] args) {

        List list1 =  new ArrayList();

        //Adding values to List using the Collection interface method
        list1.add("Jack");
        list1.add("Andrew");
        list1.add("Eric");

        System.out.println("List is: "+list1);

        //Adding values to List using the List specific method
        list1.add(1,"Jerry");

        System.out.println("\nList after insertion at index 1 is: "+list1);
    }

}
```

```java
package demo1;

import java.util.ArrayList;

public class Demo2 {
    public static void main(String[] args) {

        List list1 =  new ArrayList();
        list1.add("Jack");
        list1.add("Andrew");
        list1.add("Eric");

        System.out.println("List is: "+list1);

        //Getting the value present at an Index
        String valueAtIndex=(String)list1.get(1);
        System.out.println("\nValue at index 1 is: "+valueAtIndex);

        //Deleting the element from the List using the index
        String valueRemovedFromIndex = (String)list1.remove(2);
        System.out.println("\nValue deleted from index 2 is: "+valueRemovedFromIndex);

        System.out.println("\nList after deleting the value at the index 2 is: " +list1);

        //Checking if list is containing a value
        System.out.println("\nIs List containing Jacob: " +list1.contains("Jacob"));
        System.out.println("\nIs List containing Jack: " +list1.contains("Jack"));
    }

}
```

```java
package demo1;

import java.util.ArrayList;

public class Demo3 {
    public static void main(String[] args) {

        List list1 =  new ArrayList();
        list1.add("Jack");
        list1.add("Andrew");
        list1.add("Eric");

        List list2 =  new ArrayList();
        list2.add("Julius");
        list2.add("Dan");


        System.out.println("List1 before insertion of List 2 elements, is: "+list1);
        System.out.println("\nList2 is: "+list2);

        //Adding elements of one List to another list, using the methods of Collection interface
        list1.addAll(list2);

        System.out.println("\nList1 after insertion of List 2 elements, is: "+list1);

    }

}
```

```java
List list1 =  new ArrayList();
list1.add("Jack");
list1.add("Andrew");
list1.add("Eric");

//Iterating the list using for loop
System.out.println("Iterated Using for Loop");
System.out.println("=======================");
for(int i=0; i<list1.size();i++){
    System.out.print(list1.get(i)+"\t");
}


//Iterating the list using enhanced for loop
System.out.println("\n\nIterated Using Enhanced For Loop");
System.out.println("================================");
for(Object i:list1){
    System.out.print(i+"\t");
}

//Iterating the list using iterator
System.out.println("\n\nIterated Using Iterator");
System.out.println("=======================");
Iterator itr = list1.iterator();
while (itr.hasNext()) {
    String val = (String) itr.next();
    System.out.print(val+"\t");
}


List list1 =  new ArrayList();
list1.add("Jack");
list1.add("Andrew");
list1.add("Eric");

//Iterating using ListIterator given by List Interface
ListIterator listItr = list1.listIterator();

System.out.println("Iterated in forward direction using a List Iterator");
System.out.println("====================================================");

while (listItr.hasNext()) {
    String val = (String) listItr.next();
    System.out.print(val+"\t");
}


//Iterating in the reverse direction
System.out.println("\n\nIterated in reverse direction using a List Iterator");
System.out.println("====================================================");

while (listItr.hasPrevious()) {
    String val = (String) listItr.previous();
    System.out.print(val+"\t");
}

}
```

LinkedList:

# LinkedList

- It allows sequential access unlike array list hence slow access
- Adding and removing element is easier and faster
- It has following important additional methods:

| Name | Description |
|---|---|
| void addFirst(Object o) | Adds the specified object in the beginning of the list |
| void addLast(Object o) | Adds the specified object at the end of the list |
| Object getFirst() | Returns the first object of the list |
| Object getLast() | Returns the last object of the list |
| Object removeFirst() | Removes and returns the first object from the list |
| Object removeLast() | Removes and returns the last object from the list |

```java
package demo2;

import java.util.LinkedList;

public class Demo1 {
    public static void main(String[] args) {
        LinkedList list = new LinkedList();

        //Adding elements using the Collection interface method:
        list.add("Jack");
        list.add("Andrew");
        list.add("Eric");
        System.out.println("List is: "+list);

        //Adding values using the LinkedList method
        list.addFirst("Peter");
        System.out.println("\n\nAfter adding the value in List using addFirst(): "+list);

        //Adding elements using the LinkedList method
        list.addLast("Jerry");
        System.out.println("\n\nAfter adding the value in List using addLast(): "+list);

    }

}
```

```java
LinkedList list = new LinkedList();

//Adding elements using the Collection interface method
list.add("Jack");
list.add("Andrew");
list.add("Eric");
list.add("Jerry");
list.add("Mark");
list.add("Chandler");
System.out.println("List is : "+list);

//Getting the first value from LinkedList
String valueAtFirst = (String)list.getFirst();
System.out.println("\nGetting the first value from List using getFirst(): "+valueAtFirst);

//Getting the last value from LinkedList
String valueAtEnd = (String)list.getLast();
System.out.println("\nGetting the last value from List using getLast(): "+valueAtEnd);

//Deleting the first value from LinkedList
String valueRemovedFromFirst = (String)list.removeFirst();
System.out.println("\nRemoving the value from index 0 using removeFirst(): "+valueRemovedFromFirst);
System.out.println("List after removing value from index 0: "+list);

//Deleting the last value from LinkedList
String valueRemovedFromEnd = (String)list.removeLast();
System.out.println("\nRemoving the value from last index  using removeLast(): "+valueRemovedFromEnd);
System.out.println("List after removing value from last index: "+list);
}
```

# ArrayList vs LinkedList

| ArrayList | LinkedList |
|---|---|
| It is the array implementation of the List Interface | It is the linked list implementation of the List interface |
| Allows random access. Hence provides fast access | Allows sequential access of the list. Hence access is relatively slow |
| Adding and removing an element is time consuming | Adding and removing an element is relatively fast |

## Set Interface and Implementation classes


## Set Interface and Implementation classes

| Method Name | Example | Description |
|---|---|---|
| add() | set.add(2); | Adds an object to the collection |
| contains( ) | set.contains(4); | Returns true if a specified object is an element within the collection |
| iterator( ) | set.iterator(); | Returns an Iterator object for the collection which may be used to retrieve an object |
| remove( ) | set.remove(); | Removes a specified object from the collection. |
| size( ) | set.size(); | Returns the number of elements in the collection |
| clear( ) | set.clear(); | Removes all objects from the collection |
| isEmpty( ) | set.isEmpty(); | Returns true if the collection has no elements |

```
Set<Integer> set=new HashSet<Integer>();
```

```java
package com.demo;

import java.util.HashSet;

public class HashSetTester {

    public static void main(String[] args) {

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

        set.add(new Integer(10));
        set.add(new Integer(30));
        set.add(new Integer(20));
        set.add(new Integer(10));

        System.out.println(set);

    }

}
```

```java
public class HashSetTester {

    public static void main(String[] args) {

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

        set.add(new Integer(10));
        set.add(new Integer(30));
        set.add(new Integer(20));
        set.add(new Integer(10));

        System.out.println(set);

        Iterator<Integer> iterator = set.iterator();
        System.out.println("Displaying set  using Iterator");
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }

    }

}
```

```java
System.out.println("set.contains(110) gives: "+set.contains(110));

set.remove(10);
System.out.println("Contents of set after removing 10: "+set);
```

```java
System.out.println("Checking if set is Empty or not: "+set.isEmpty());
```

```java
package com.demo;

import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;

public class LinkedHashSetTester {

    public static void main(String[] args) {
        Set<Integer> set = new LinkedHashSet<Integer>();
        set.add(new Integer(10));
        set.add(new Integer(30));
        set.add(new Integer(20));
        set.add(new Integer(10));

        Iterator<Integer> iterator = set.iterator();
        System.out.println("Displaying LinkedHashSet  using Iterator");
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
```

```
Displaying LinkedHashSet  using Iterator
10
30
20
```

Set cannot contain duplicate elements

LinkedHashSet : Order of elements will be in insertion order

```java
package com.demo;

import java.util.Iterator;

public class TreeSetTester {

    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        set.add(new Integer(10));
        set.add(new Integer(30));
        set.add(new Integer(20));
        set.add(new Integer(10));

        Iterator<Integer> iterator = set.iterator();
        System.out.println("Displaying TreeSet using Iterator");
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }

}
```

```
Displaying TreeSet using Iterator
10
20
30
```

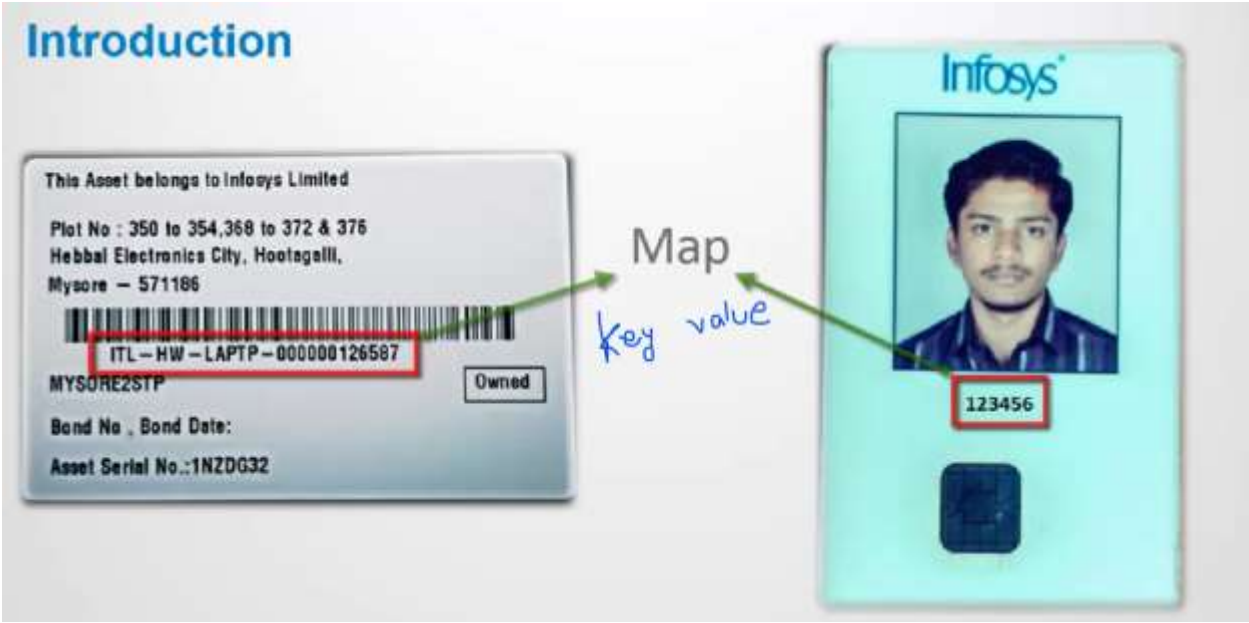
Set cannot contain duplicate elements

TreeSet : Order of elements will be in sorted order

- Set is a collection that cannot contain duplicate elements
- HashSet, LinkedHashSet, TreeSet
- Some important methods:
  - add()
  - contains()
  - iterator()
  - remove()
  - size()
  - clear()
  - isEmpty()

# Map Interface

It is an interface which maps unique keys to values. These unique keys are used to retrieve the values


Introduction
This Asset belongs to Infosys Limited
Plot No : 350 to 354,368 to 372 & 376
Hebbal Electronics City, Hootagalli,
Mysore – 571186
ITL–HW–LAPTP–000000126587
MYSORE2STP
Owned
Bond No , Bond Date:
Asset Serial No.:1NZDG32
Map
Key value
Infosys
123456


Map
HashMap
No guarantee to the order of the elements
LinkedHashMap
The insertion order is maintained
TreeMap
Elements are ordered based on their sorted ordering of the keys

Map<Integer, String> map = new HashMap<Integer, String>  map.put(1001, "Josh")

**map**

| Method Name | Example | Description |
|---|---|---|
| put() | map.put(1001, "Josh") | Adds an object in the form of a key and value pair to the collection. |
| remove() | map.remove(1001); | Returns the corresponding value that has been removed using the key. |
| containsKey() | map.containsKey(1001); | Returns true if the key passed as a parameter is present in the collection |
| containsValue() | map.containsValue("Josh"); | Returns true if the value passed as a parameter is present in the collection. |
| isEmpty() | map.isEmpty(); | Returns true if the map is empty, i.e.; if it doesn't contain any elements. |
| keySet() | map.keySet(); | Returns the set of all the keys present inside the map. |
| size() | map.size(); | Returns an integer which is the no of elements in the map. |

```java
package com.demo;

import java.util.HashMap;
import java.util.Map;

public class HashMapTester {

    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(new Integer(10), "Hello");
        map.put(new Integer(30), "Welcome");
        map.put(new Integer(20), "Ok");
        map.put(new Integer(20), "Test");
        System.out.println("Displaying key-value pairs:"+map);

    }
}
```

Map can contain only unique keys, adding a duplicate key with a value will simply overrides the existing value.

# HashMap: Elements are displayed in random order

```java
public class HashMapTester {

    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(new Integer(10), "Hello");
        map.put(new Integer(30), "Welcome");
        map.put(new Integer(20), "Ok");
        map.put(new Integer(20), "Test");
        System.out.println("Displaying key-value pairs:"+map);

        System.out.println("\nDisplaying keys of a map:");
        Set<Integer> set = map.keySet();
        for(Integer i:set) {
            System.out.println(map.get(i));
        }

        map.remove(10);
        System.out.println("\nDisplaying Key-value pairs:"+map);

    }
}
```

```java
System.out.println("\nKey 10 is present in the map:"+map.containsKey(10));

System.out.println("\nThe value Test is present in the map:"+map.containsValue("Test"));

System.out.println("\nThere are no elements in the map:"+map.isEmpty());

System.out.println("\nThe size of the map is:"+map.size());
```

```java
package com.demo;

import java.util.LinkedHashMap;

public class LinkedHashMapTester {

    public static void main(String[] args) {

        Map<Integer, String> map = new LinkedHashMap<Integer, String>();
        map.put(new Integer(10), "Hello");
        map.put(new Integer(30), "Welcome");
        map.put(new Integer(20), "Ok");
        map.put(new Integer(20), "Test");

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

LinkedHashMap: Elements are displayed in insertion order

```java
package com.demo;

import java.util.Map;

public class TreeMapTester {

    public static void main(String[] args) {

        Map<Integer, String> map = new TreeMap<Integer, String>();
        map.put(new Integer(10), "Hello");
        map.put(new Integer(30), "Welcome");
        map.put(new Integer(20), "Ok");
        map.put(new Integer(20), "Test");

        System.out.println(map);

        }
    }
```

```
{10=Hello, 20=Test, 30=Welcome}
```

TreeMap: Elements are displayed in sorted order of keys

To see difference between hashmap and concurrentHashmap lets first create hashmap and remove elements

```java
package com.demo;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class ConcurrentHashMapTester {

    public static void main(String[] args) {

        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(new Integer(10), "Hello");
        map.put(new Integer(30), "Welcome");
        map.put(new Integer(20), "Ok");

        Set<Integer> set = map.keySet();
        for (Integer i : set) {
            map.remove(i);
        }

        System.out.println(map.size());
    }
```

We will get exception

ConcurrentModification Exception: Because we are trying to modify the contents while iterating the same

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(HashMap.java:926)
        at java.util.HashMap$KeyIterator.next(HashMap.java:960)
        at com.demo.ConcurrentHashMapTester.main(ConcurrentHashMapTester.java:17)
```

Thus we can use ConcurrentHashMap

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapTester {

    public static void main(String[] args) {

        Map<Integer, String> map = new ConcurrentHashMap<Integer, String>();
        map.put(new Integer(10), "Hello");
        map.put(new Integer(30), "Welcome");
        map.put(new Integer(20), "Ok");
```

```
0
```

HashMap is non synchronized, and not thread safe and cannot be used for concurrent multi-threaded environment

ConcurrentHashMap is synchronized and thread safe

- Map stores the elements in a key-value pair
- HashMap, LinkedHashMap, TreeMap and ConcurrentHashMap
- Some important methods:
  - put()
  - remove()
  - containsKey()
  - containsValue()
  - isEmpty()
  - size()

Regular Expressions:

## Introduction

```java
String passportNumber="L1234567";
int flag= 0;

if(passportNumber.length()<8){
    System.out.println("Invalid Passport Number");
}

for(int i=0; i<passportNumber.length();i++)
{
    char ch = passportNumber.charAt(i);

    if(!((ch>='A' && ch<='Z')
       ||(ch>='a' && ch<='z')
       ||(ch>='0' && ch<='9'))
    )
    {
        flag=1;
        break;
    }
}

if(flag==0){
    System.out.println("Valid Passport Number");
}
else{
    System.out.println("Invalid Passport Number");
}
```

```java
String passportNumber="L1234567";
boolean flag= false;

String patt="[a-zA-Z0-9]{8,}";

flag=passportNumber.matches(patt);
if(flag){
    System.out.println("Valid Passport Number");
}
else{
    System.out.println("Invalid Passport Number");
}
```

## Regular Expressions (regex) in Java

- Regular Expression (**regex**) is a sequence of characters that forms a search pattern
- The search pattern is used in searching and editing a String
- **regex** was added to Java from JDK 1.4 under the package java.util.regex
- Regular expression provides the following options:
  - Character Classes
  - Quantifiers
  - Meta Character
- The String class contains the following methods that support regex:
  - matches(), split(), replaceFirst(), replaceAll()

# Character Classes

- Character classes are used to match a character in a String
- Character classes define what all characters a String may contain
- [] is used to represent a character class

| Character Class | Description |
|---|---|
| . | Matches any character |
| [abc] | A String can contain any character from a, b or c |
| [abc][vz] | A String can contain any character from a, b or c, followed by v or z |
| [a-z] | A String can contain any character from range of a to z |
| [a-zA-Z] | A String can contain any character from range of a to z and A to Z |
| [a-d1-7] | A String can contain any character from range of a to d and digits from a range of 1 to 7 |
| A\|B | A String can contain any character either A or B |
| [^abc] | A String can contain any character apart from a, b and c |

```java
package demo1;


public class Demo1 {


    public static void main(String[] args) {

        // String should be numeric
        String s1 =  "1";
        String s2 =  "A";

        String pattern="[0-9]";

        System.out.println("Pattern is: "+pattern);
        System.out.println("=================================\n");
        System.out.println("Is S1 Numeric: "+s1.matches(pattern));
        System.out.println("Is S2 Numeric: "+s2.matches(pattern));

    }

}
```

true

```java
package demo1;

public class Demo2 {

    public static void main(String[] args) {

        // String should contain only alphabets from A to Z or a-z
        String s1 =  "A";
        String s2 =  "b";
        String s3 =  "1";

        String pattern = "[a-zA-Z]";

        System.out.println("Pattern is: " +pattern);
        System.out.println("=========================================\n");
        System.out.println("Is S1 an alphabet  "+s1.matches(pattern));
        System.out.println("Is S2 an alphabet  "+s2.matches(pattern));
        System.out.println("Is S3 an alphabet  "+s3.matches(pattern));

    }

}
```

```java
package demo1;

public class Demo3 {

    public static void main(String[] args) {

        // String should contain only alphabets from A to Z or a-z
        String s1 =  "Aa";
        String s2 =  "bA";
        String s3 =  "1";

        String pattern = "[a-zA-Z]";

        System.out.println("Pattern is: " +pattern);
        System.out.println("=========================================\n");
        System.out.println("Is S1 an alphabet: "+s1.matches(pattern));
        System.out.println("Is S2 an alphabet: "+s2.matches(pattern));
        System.out.println("Is S3 an alphabet: "+s3.matches(pattern));

    }

}
```

Valid characters

Validates strings of Length 1

# Quantifiers

- Quantifiers define how many times a character can exist in a String

| regex | Description |
|---|---|
| X{n} | In a String, X occurs strictly **n times** |
| X{n,} | In a String, X occurs **n or more times** |
| X{m,n} | In a String, X occurs a minimum of **m times** and a maximum of **n times** |
| X? | In a String, X occurs zero or once |
| X+ | In a String, X occurs one or more times. It is equivalent to X{1,n} |
| X* | In a String, X occurs zero or more times. It is equivalent to X{0,n} |

```java
package demo2;

public class Demo1 {

    public static void main(String[] args) {

        // String should contain only alphabets from A to Z or a-z
        String s1 =  "Aa";
        String s2 =  "bA";
        String s3 =  "1";

        String pattern = "[a-zA-Z]+";

        System.out.println("Pattern is: " +pattern);
        System.out.println("=========================================\n");
        System.out.println("Is S1 an alphabet: "+s1.matches(pattern));
        System.out.println("Is S2 an alphabet: "+s2.matches(pattern));
        System.out.println("Is S3 an alphabet: "+s3.matches(pattern));

    }

}
```

```java
package demo2;

public class Demo2 {

    public static void main(String[] args) {

        // String should contain only alphabets from A to Z or a-z
        String s1 =  "Apple";
        String s2 =  "bAd";
        String s3 =  "Oranges";

        String pattern = "[a-zA-Z]{2,5}";

        System.out.println("Pattern is: " +pattern);
        System.out.println("==========================================\n");
        System.out.println("Is S1 having length between 2 and 5: "+s1.matches(pattern));
        System.out.println("Is S2 having length between 2 and 5: "+s2.matches(pattern));
        System.out.println("Is S3 having length between 2 and 5: "+s3.matches(pattern));

    }

}
```

# Meta Characters

- Meta Characters are a predefined set of patterns

| Meta Characters | Description |
|---|---|
| \d | A digit character, equivalent to [0-9] |
| \D | A non digit character, equivalent to [^0-9] |
| \s | A white space character |
| \S | A non white space character |
| \w | A word character, that may contain any character from a to z, A to Z, underscore("_") or 0 to 9] |
| \W | A non word character |

```java
package demo3;

public class Demo1 {

    public static void main(String[] args) {

        // String should contain only alphabets from A to Z or a-z
        String s1 =  "Apple";
        String s2 =  "bA8d";
        String s3 =  "Oranges";

        String pattern = "\\w(2,5)";

        System.out.println("Pattern is: " +pattern);
        System.out.println("=========================================\n");
        System.out.println("Is S1 having length between 2 and 5: "+s1.matches(pattern));
        System.out.println("Is S2 having length between 2 and 5: "+s2.matches(pattern));
        System.out.println("Is S3 having length between 2 and 5: "+s3.matches(pattern));

    }

}
```

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class UserInterface {
    public boolean validateMobileNumber(String mobileNumber) {
        Pattern regex = Pattern.compile("\\d{3}-\\d{3}-\\d{4}");
        Matcher mobileMatcher = regex.matcher(mobileNumber);
        return mobileMatcher.matches();
    }

    public static void main(String[] args) {
        UserInterface object = new UserInterface();
        System.out.println(object.validateMobileNumber("111-222-3333"));
    }
}
```

Data Persistence:

# Persistence in Applications

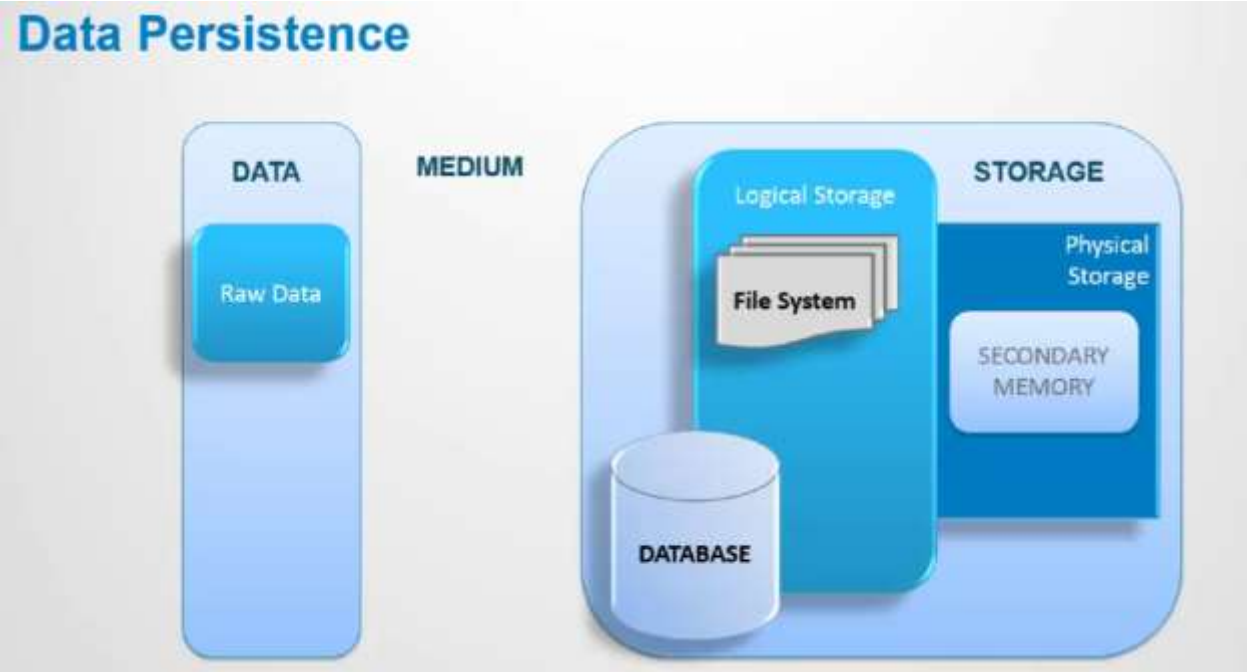
- Permanent storage of data is called as persistence and it is one of the most critical operations in an application
- All the systems or applications need to persist data in some form depending on the specific requirements

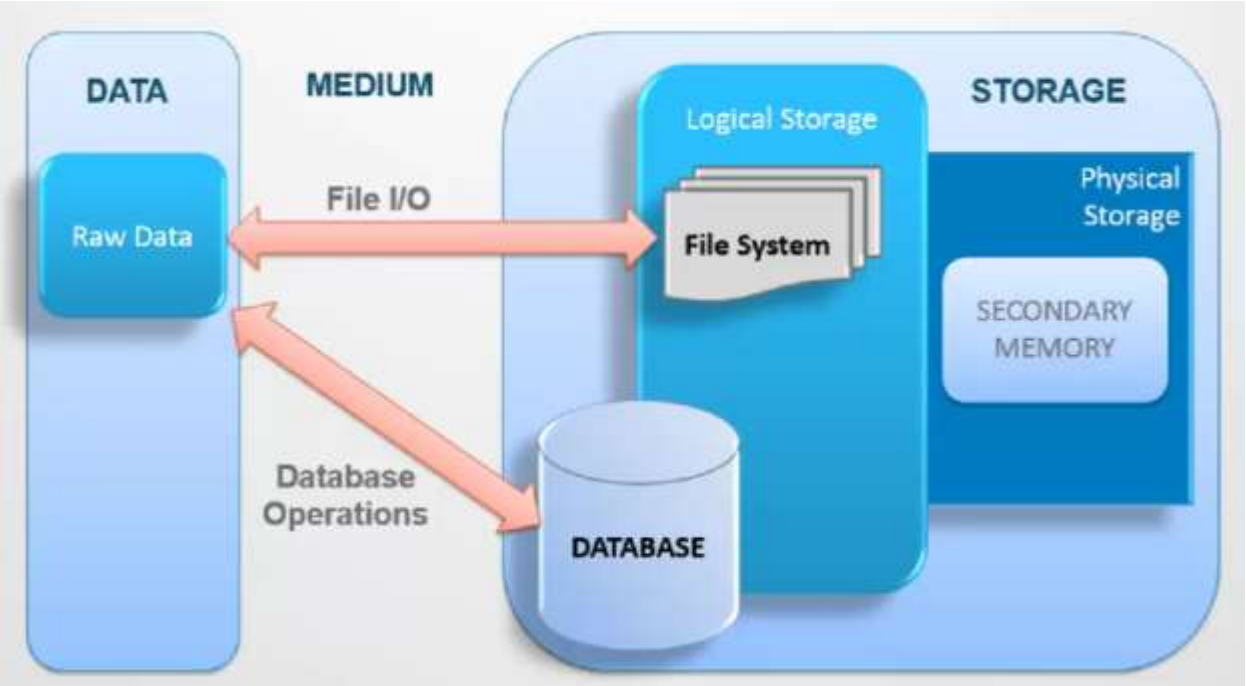

# Components of Persistence

- The Persistence phenomenon consists of three core components
  - Data
  - Medium
  - Storagees


,

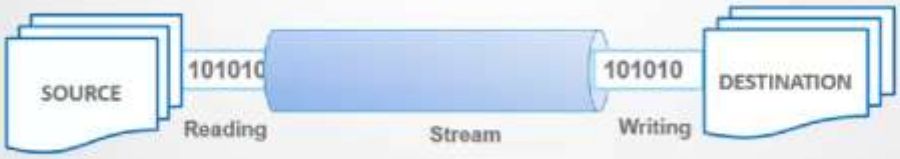
- In order to implement Persistence for an Enterprise Application the details of these core components are crucial




File Input Output Streams in Java:

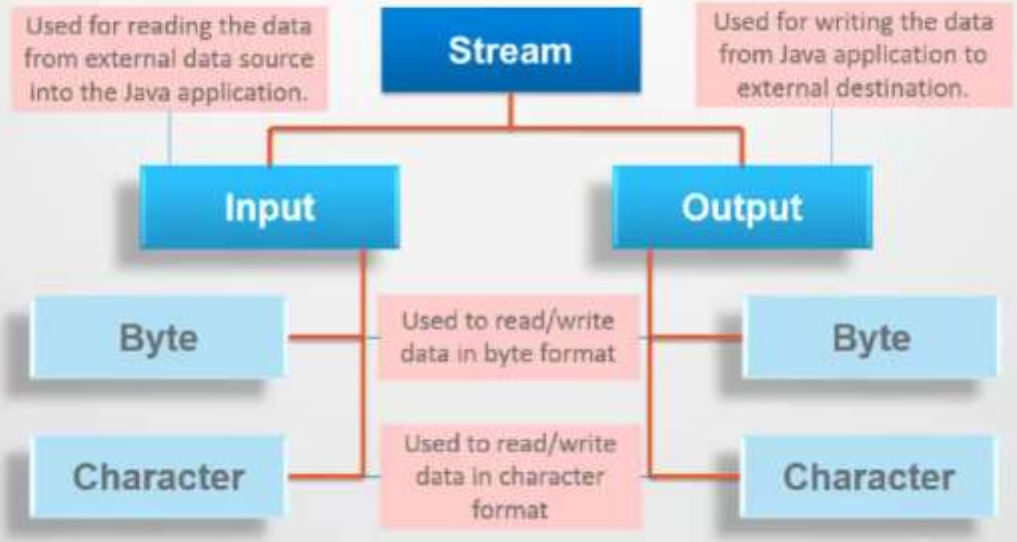
# Java I/O Streams




- A stream is a sequence of data
- Streams support different types of data, e.g. simple bytes, primitive data types, etc.
- Streams are used to read input data from the source and write output data to the destination
- The source and destination can be anything that holds, generates, or consumes data, e.g. File System, application program, peripheral devices etc.

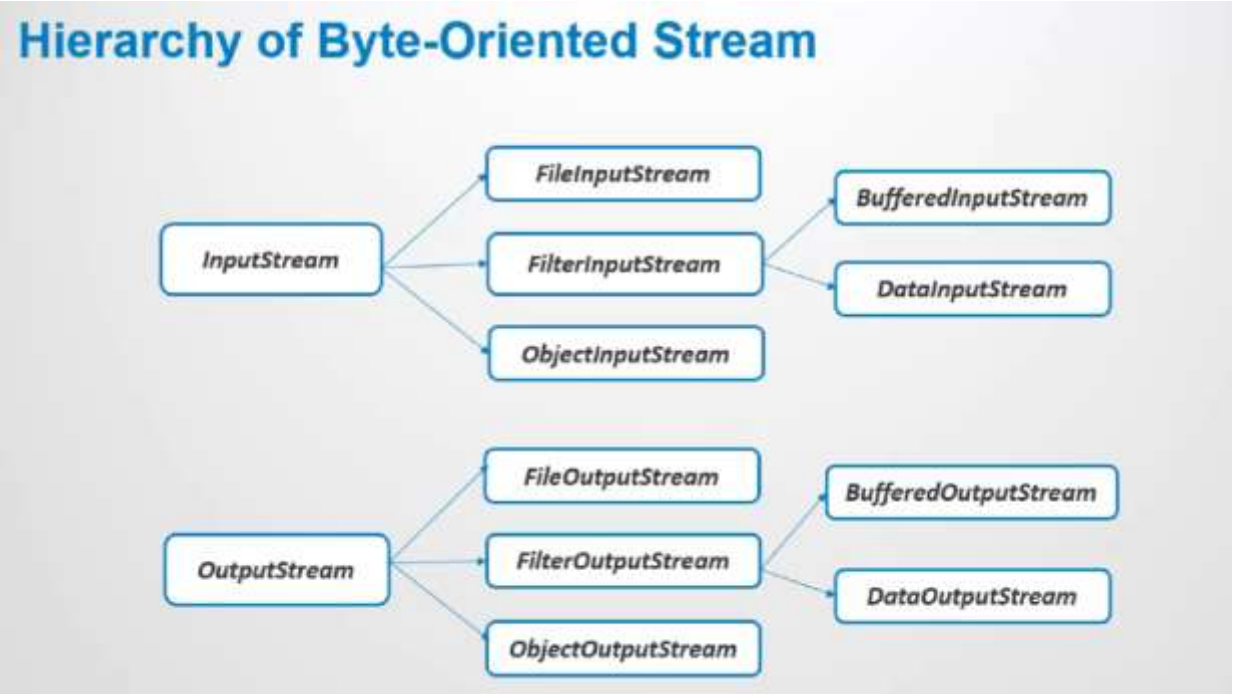
**Implementation of Java I/O Streams is available in *java.io* package**

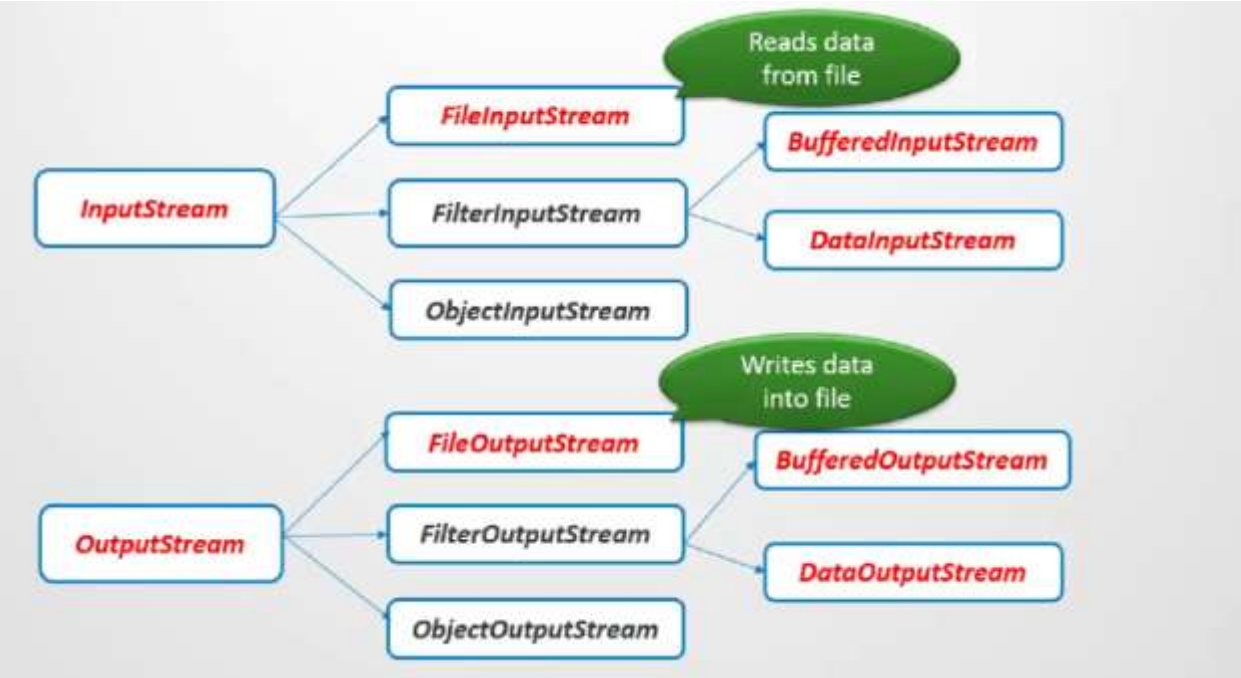
# Types of Streams

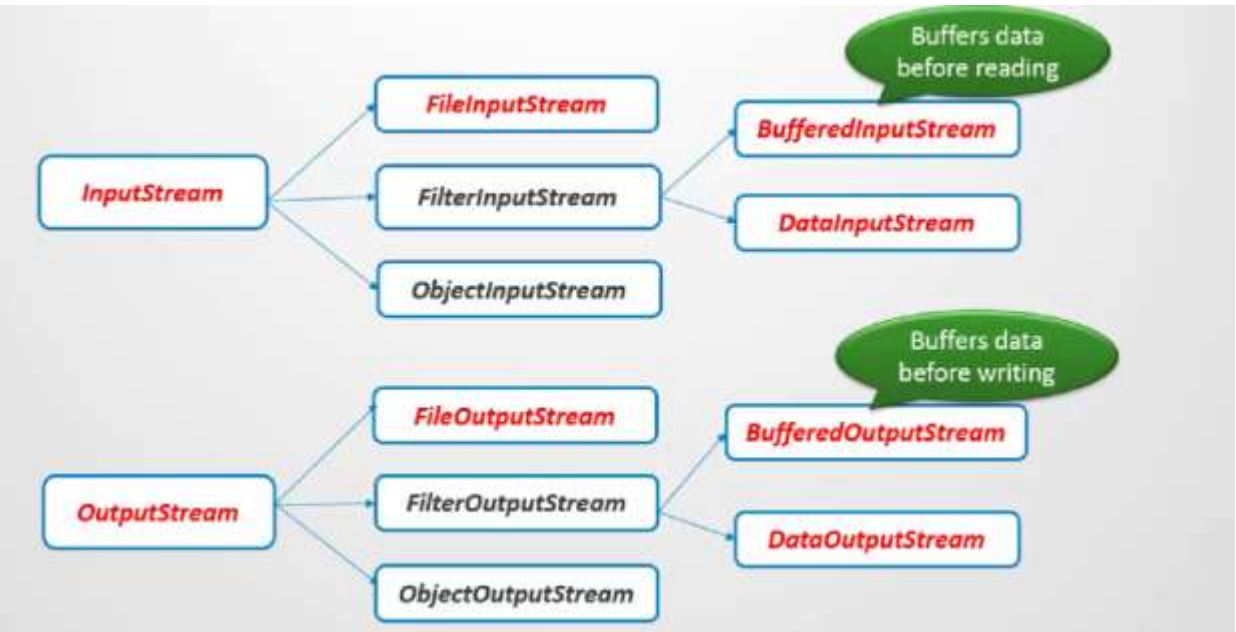


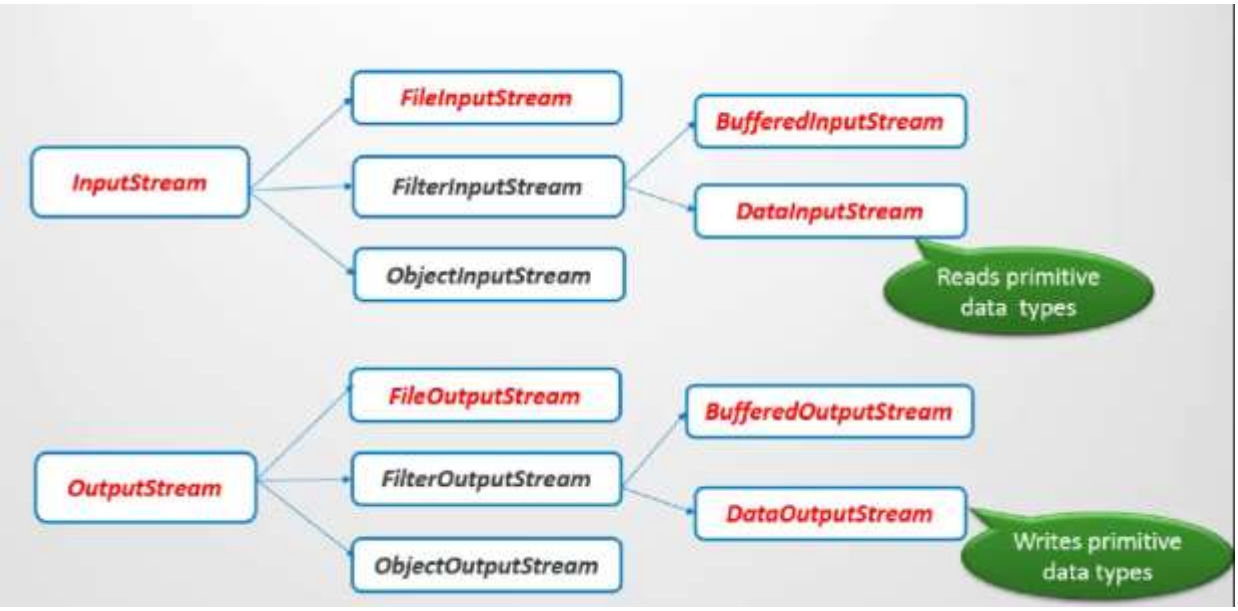

# Byte-Oriented Streams

- Byte Streams read data from the source and write data to the destination in the form of bytes (8 bits)
- All Byte Stream classes are derived from the following two abstract classes
  - java.io.InputStream: Used for reading data in the form of bytes
  - java.io.OutputStream: Used for writing data in the form of bytes


Hierarchy of Byte-Oriented Stream
InputStream
FileInputStream
FilterInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
OutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
BufferedOutputStream
DataOutputStream


Reads data from file
InputStream
FileInputStream
FilterInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
Writes data into file
OutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
BufferedOutputStream
DataOutputStream


Buffers data before reading
InputStream
FileInputStream
FilterInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
Buffers data before writing
OutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
BufferedOutputStream
DataOutputStream

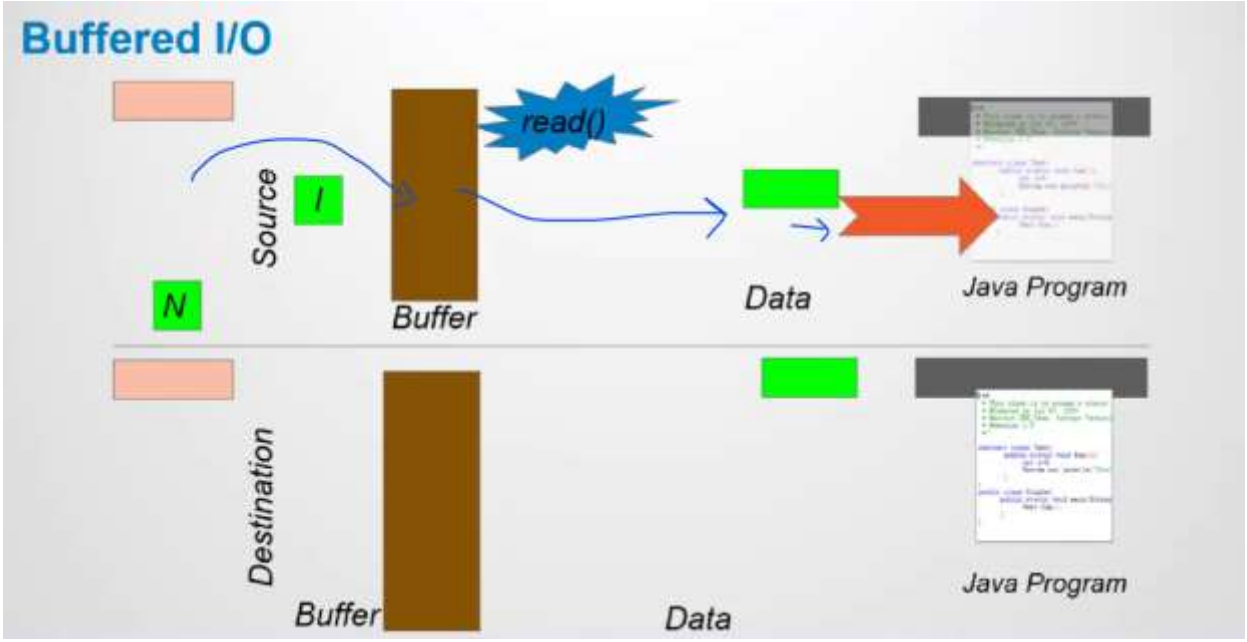

InputStream
FileInputStream
FilterInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
Reads primitive data types
OutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
BufferedOutputStream
DataOutputStream
Writes primitive data types

```java
package javalangfeatures;

import java.io.FileNotFoundException;

public class FileOutputStreamDemo {

    public static void main(String[] args) throws IOException {
        try {
            //if we give just filename as Data.dat file will be created under project folder
            //if we give path of file then it will use that file
            FileOutputStream fos = new FileOutputStream("Data.dat");
//          FileOutputStream fos = new FileOutputStream("C:\\Testing\\SeleniumJava\\00-Selenium\\JavaLangFeatures\\src\\resources\\data")
            String value = "Java is Fun to Learn loving it";

            byte[] val = value.getBytes();

            try {
                fos.write(val);
                fos.write(val[0]);
            } catch (IOException e) {

                e.printStackTrace();
            } finally {

                fos.close();
            }

        } catch (FileNotFoundException e) {

            System.out.println(e.getMessage());
        }
        System.out.println("Data written Successfully");
    }

}
```

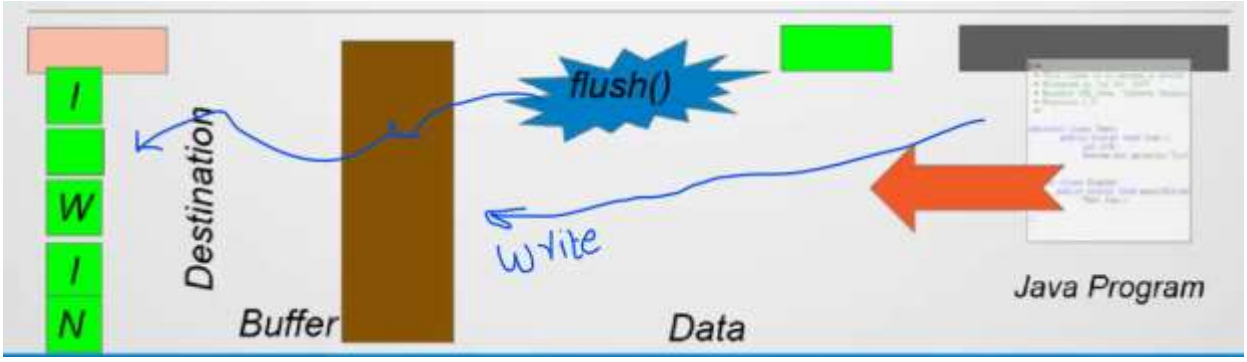
FileInputSTreamDemo.java × data

```java
package javalangfeatures;

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

public class FileInputSTreamDemo {

    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream(
                    "C:\\Testing\\SeleniumJava\\00-Selenium\\JavaLangFeatures\\src\\resources\\data");

            System.out.println("Data in file is :");

            int i = fis.read();

            // checking for End of File
            while (i != -1) {
                System.out.print((char) i);
                i = fis.read();
            }

            // closing the stream
            fis.close();

        } catch (FileNotFoundException e) {

            e.printStackTrace();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }

    }

}
```

Buffered I/O




Chunks of data will be passed from source to buffer and then it will be passed to program, while reading file if it reaches EOF it will return -1




With write method data from java program is sent to buffer and then when flush method is invoked whole data from buffer goes to destination

```java
public class BufferOutputStreamTester {

    public static void main(String[] args) {

        try
        {                                   // the file name in which data has to be written
            FileOutputStream fos = new FileOutputStream("Data.dat");

            // stream chaining
            BufferedOutputStream bos = new BufferedOutputStream(fos);

            //Content to be written
            String value ="Happy to learn Java Buffered I/O";

            byte[] byteArray= value.getBytes();

            // writing the content
            bos.write(byteArray);

            bos.write(byteArray[0]);

            //closing the stream
            bos.close(); // internally calls flush()

            System.out.println("Data written to file successfully");
        }
        catch (FileNotFoundException exception) {
            System.out.println(exception.getMessage());
        }
        catch (IOException exception) {
```

Stream Chaining

```java
public class BufferOutputStreamTester {

    public static void main(String[] args) {

        try
        {                                   // the file name in which data has to be written
            FileOutputStream fos = new FileOutputStream("Data.dat");

            // stream chaining
            BufferedOutputStream bos = new BufferedOutputStream(fos);

            //Content to be written
            String value ="Happy to learn Java Buffered I/O";

            byte[] byteArray= value.getBytes();

            // writing the content
            bos.write(byteArray);

            bos.write(byteArray[0]);

            //closing the stream
            bos.close(); // internally calls flush()

            System.out.println("Data written to file successfully");
        }
        catch (FileNotFoundException exception) {
            System.out.println(exception.getMessage());
        }
        catch (IOException exception) {
```

Using close() will close the chained stream as well..

Using close() will invoke the flush() method Implicitly

```java
10    public static void main(String[] args) {
11        try
12        {
13                                // the file name from which data has to be read
14            FileInputStream fis = new FileInputStream("Data.dat");
15
16            // stream chaining
17            BufferedInputStream bis = new BufferedInputStream(fis);
18
19            System.out.println("Data in the file is:");
20
21            int i = bis.read();
22
23            //checking for end of file
24            while (i != -1)
25            {
26                //printing and reading the content from the file
27                System.out.print((char) i);
28                i = bis.read();
29            }
30
31            //closing the stream
32            bis.close();
33
34        }catch (FileNotFoundException exception) {
35            System.out.println(exception.getMessage());
36        }
37        catch (IOException exception) {
38            System.out.println(exception.getMessage());
39        }
```

```java
10    public static void main(String[] args) {
11        try
12        {
13                                // the file name from which data has to be read
14            FileInputStream fis = new FileInputStream("Data.dat");
15
16            // stream chaining
17            BufferedInputStream bis = new BufferedInputStream(fis);
18
19            System.out.println("Data in the file is:");
20
21            int i = bis.read();
22
23            //checking for end of file
24            while (i != -1)
25            {
26                //printing and reading the content from the file
27                System.out.print((char) i);
28                i = bis.read();
29            }
30
31            //closing the stream
32            bis.close();
33
34        }catch (FileNotFoundException exception) {
35            System.out.println(exception.getMessage());
36        }
37        catch (IOException exception) {
38            System.out.println(exception.getMessage());
39        }
```

Open the Stream(s)

Perform the required Operations

Close the Stream

Usage of try with resources:

```java
10    public static void main(String[] args) {
11
12        //Usage of try with resources
13        try
14        {
15         FileOutputStream fos = new FileOutputStream("Data.dat"); // file name in which data has to be written
16         BufferedOutputStream bos = new BufferedOutputStream(fos) // stream chaining
17        }
18
19        {
20            //Content to be written
21            String value ="Happy to learn Java Buffered I/O";
22
23            byte[] byteArray= value.getBytes();
24
25            // writing the content
26            bos.write(byteArray);
27
28            bos.write(byteArray[0]);
29
30            //closing the stream
31            //bos.close(); // internally calls flush()
32
33            System.out.println("Data written to file successfully");
34        }
35        catch (FileNotFoundException exception) {
36            System.out.println(exception.getMessage());
37        }
38        catch (IOException exception) {
39            System.out.println(exception.getMessage());
```

This is available from java 7 version , using this the file will be closed automatically

DataOutputStream : reads and writes any type of data

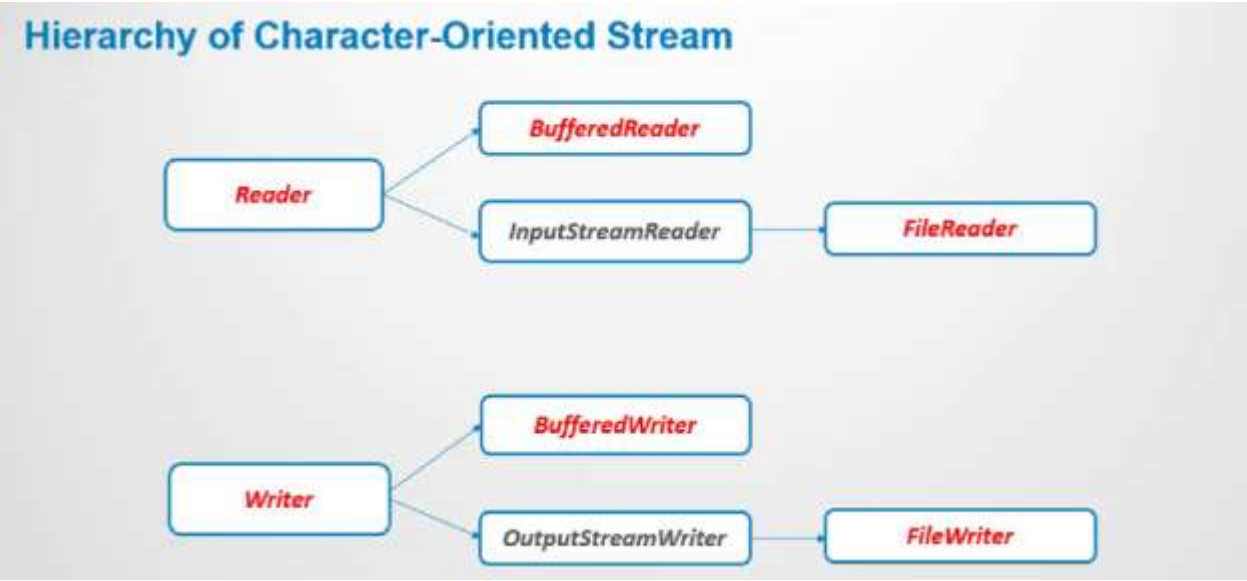
```java
7
8  public class DataOutputStreamTester {
9
10    public static void main(String[] args) {
11        try
12        {
13            FileOutputStream fos = new FileOutputStream("Data.dat"); //the file name in which data has to be written
14            DataOutputStream dos = new DataOutputStream(fos);  // stream chaining
15        }
16
17        {
18
19            //Writing the content in the file
20            dos.writeInt(100);
21            dos.writeDouble(56.78);
22            dos.writeFloat(4.5f);
23
24            System.out.println("Data written successfully");
25        }
26        catch (FileNotFoundException exception) {
27            System.out.println(exception.getMessage());
28        }
29        catch (IOException exception) {
30            System.out.println(exception.getMessage());
31        }
32        catch (Exception exception) {
33            System.out.println(exception.getMessage());
34        }
35    }
```

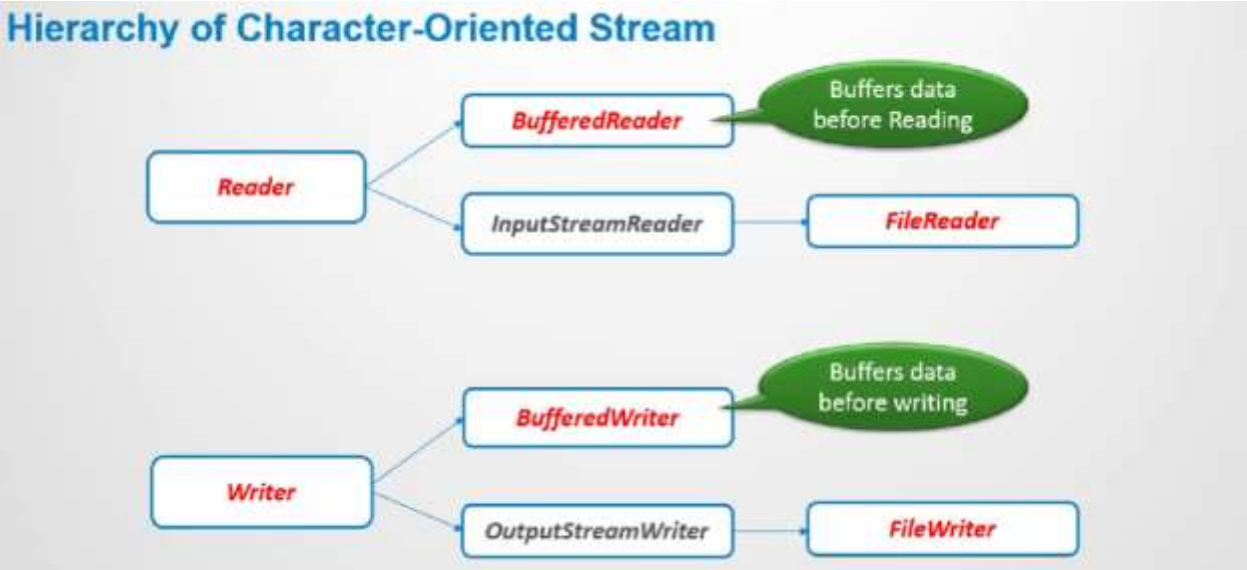
```java
public class DataOutputStreamTester {

    public static void main(String[] args) {
        try
        {
                FileOutputStream fos = new FileOutputStream("Data.dat");//the file name in which data has to be written
                DataOutputStream dos = new DataOutputStream(fos);  // stream chaining
        }
        {
            //Writing the content in the file
            dos.writeInt(100);
            dos.writeDouble(56.78);
            dos.writeFloat(4.5f);

            System.out.println("Data written successfully");
        }
        catch (FileNotFoundException exception) {
            System.out.println(exception.getMessage());
        }
        catch (IOException exception) {
            System.out.println(exception.getMessage());
        }
        catch (Exception exception) {
            System.out.println(exception.getMessage());
        }
    }

    public static void main(String[] args) {
        try
        {
                FileInputStream fis = new FileInputStream("Data.dat"); // the file name from which data has to be read
                DataInputStream dis = new DataInputStream(fis); // stream chaining
        }
        {
            // reading the content
            int i  = dis.readInt();
            double d = dis.readDouble();
            float f = dis.readFloat();

            //displaying the content
            System.out.println("Data in the file is:");
            System.out.println(i);
            System.out.println(d);
            System.out.println(f);
        }
        catch (FileNotFoundException exception) {
            System.out.println(exception.getMessage());
        }
        catch (IOException exception) {
            System.out.println(exception.getMessage());
        }
        catch (Exception exception) {
            System.out.println(exception.getMessage());
        }
    }
```

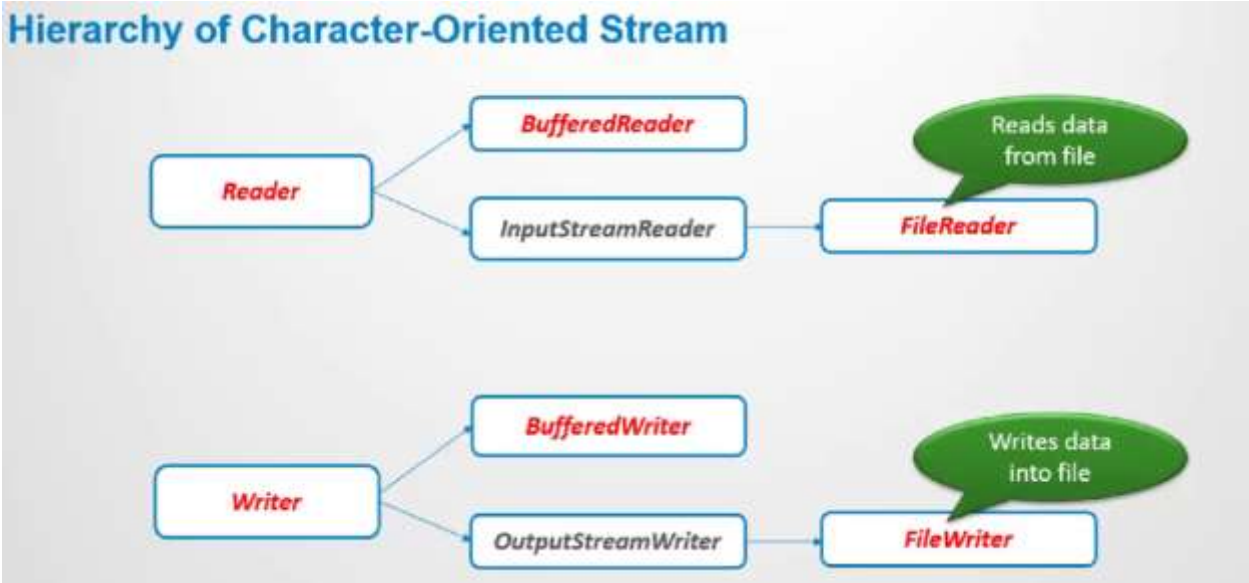
These methods are exclusive to DataOutputStream class

## Character-Oriented Stream

- Character Streams reads data from the source and writes data to the destination in the form of characters(16 bit Unicode)
- All character stream classes are derived from the following two abstract classes
  - java.io.Reader: Used for reading data in the form of characters
  - java.io.Writer: Used for writing data in the form of characters


Hierarchy of Character-Oriented Stream
Reader
BufferedReader
InputStreamReader
FileReader
Writer
BufferedWriter
OutputStreamWriter
FileWriter


Hierarchy of Character-Oriented Stream
Reader
BufferedReader
Buffers data before Reading
InputStreamReader
FileReader
Writer
BufferedWriter
Buffers data before writing
OutputStreamWriter
FileWriter




```java
public class FileWriterTester {

    public static void main(String[] args) {
        try
        {   //Creating the object for FileWriter
            FileWriter fw = new FileWriter("Data.txt");

            // writing the content
            fw.write("Learning Character Oriented Streams");

            System.out.println("Data written successfully");

            //Closing the writer
            fw.close();

        }
        catch (FileNotFoundException exception) {
            System.out.println(exception.getMessage());
        }
        catch (IOException exception) {
            System.out.println(exception.getMessage());
        }
        catch (Exception exception) {
            System.out.println(exception.getMessage());
        }
    }

}
```

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

        try
        {
            FileReader fr = new FileReader("Data.txt"); //the file name from which data has to be read
            System.out.println("Data in the file is:");

            //reading the content
            int i  = fr.read();

            while(i != -1) // to check for the end of file
            {
                System.out.print((char)i);
                i = fr.read(); // reading the content
            }

            //Closing the reader
            fr.close();
        }
        catch (FileNotFoundException exception) {
            System.out.println(exception.getMessage());
        }
        catch (IOException exception) {
            System.out.println(exception.getMessage());
        }
        catch (Exception exception) {
            System.out.println(exception.getMessage());
        }
    }
```

```java
public class BufferWriterTester {

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

            //Creating the object for FileWriter
            FileWriter fw = new FileWriter("Data.txt");
            BufferedWriter bw = new BufferedWriter(fw);  // stream chaining

            // writing the content
            bw.write("Learning BufferedWriter....");


            //Closing the writer
            bw.close();


            System.out.println("Data written to file successfully");
        }
        catch(IOException ioe) {
            System.out.println(ioe.getMessage());
        }
        catch(Exception exception) {
            System.out.println(exception.getMessage());
        }
    }

}
```

```java
public class BufferReaderTester {

    public static void main(String[] args) {
        try
        {
            FileReader fr = new FileReader("Data.txt");
            BufferedReader br = new BufferedReader(fr); // stream chaining

            System.out.println("Data in the file is:");

            // reading the content
            int i  = br.read();

            while(i != -1) {
                System.out.print((char)i);
                i = br.read();
            }

            //Closing the Reader
            br.close();

        }
        catch(IOException ioe) {
            System.out.println(ioe.getMessage());
        }
        catch(Exception exception) {
            System.out.println(exception.getMessage());
        }
    }
```

# File Class

- java.io.File is an abstract representation of the file or directory on a file system identified by a pathname
- The pathname may be
  - **Absolute** : relative to the root directory of the file system
  - **Relative** : relative pathnames against the current user directory
- File provides API for creation of files and directories, file searching, file deletion etc.

FileTesterDisplayAllFiles.java

```java
package demo3;

import java.io.File;

public class FileTesterDisplayAllFiles {

    public static void main(String[] args) {

        // Creating an Object
        File file = new File("resources/");

        // getting all the files
        File[] fileArray = file.listFiles();

        System.out.println("Following files are present under resources folder:");
        System.out.println("==================================================\n");

        // printing the file name
        for (File f : fileArray) {
            System.out.println(f.getName());
        }
    }

}
```

can be folder or filepath

Serialization and DeSerialization: