Abstract Classes

package main;

public class AbstractClasses {

// ABSTRACT PARENT CLASS

abstract static class Animal {

private String name;

public Animal(String name) {

this.name = name;

}

public String getName() {

return name;

}

/\* Abstract method, not being initialised \*/

public abstract void makeSound();

}

// CHILD CLASS INHERITING FROM ABSTRACT CLASS

static class Dog extends Animal {

public Dog(String name) {

super(name);

}

/\* Implementing the abstract method \*/

@Override

public void makeSound() {

System.out.println("Woof!");

}

}

public static void main(String[] args) {

Dog dog = new Dog("Buddy");

System.out.println(dog.getName());

dog.makeSound();

}

}

Arrays

package main;

import java.util.Arrays;

public class ArrayObjects {

public static void main(String[] args) {

// PRIMITIVE INITIALISATION

/\*array is initialised but filled later \*/

int[] array1;

array1 = new int[]{5,1,3,2,4};

intToString(array1);

/\* size of array is known but values not yet \*/

int[] array2 = new int[10];

intToString(array2);

int[] array3 = new int[2];

intToString(array3);

/\* values of array are known \*/

int[] array4 = {1, 2, 3, 4, 5};

intToString(array4);

// OBJECT INITIALISATION

Integer[] objArray = {1, 2, 3, 4, 5};

toString(objArray);

Double[] doubleObjArray = new Double[5];

toString(doubleObjArray);

// FILLING ARRAYS SEPARATELY

array3[0] = 10;

array3[1] = 20;

intToString(array3);

// FILLING ARRAYS WITH LOOPS

for (int i = 0; i < array2.length; i++) {

array2[i] = 2 \* i;

}

intToString(array2);

// ACCESSING ELEMENTS

int fourthElement = array2[3];

System.out.println(fourthElement);

// ARRAY LENGTH

System.out.println(objArray.length);

// COPY ARRAYS

/\* Using System.arraycopy() \*/

int[] copy1 = new int[array1.length];

System.arraycopy(array1, 0, copy1, 0, array1.length);

intToString(array1);

intToString(copy1);

/\* Using main.ArrayObjects.copyOf() \*/

int[] copy2 = Arrays.copyOf(array2,array2.length);

intToString(array2);

intToString(copy2);

// MULTIDIMENSIONAL ARRAYS

int[][] matrix = new int[3][3];

int[][][] cube = new int[3][3][3];;

// SORT ARRAY

Arrays.sort(array1);

intToString(array1);

// SEARCH ARRAY FOR AN ELEMENT

int index = Arrays.binarySearch(array1, 3);

System.out.println(index);

}

// ITERATING OVER ARRAYS

public static <T> void toString(T[] array) {

for (T a : array) {

System.out.print(a + " ");

}

System.out.println();

}

public static void intToString(int[] array) {

for (int a : array) {

System.out.print(a + " ");

}

System.out.println();

}

}

Attributes

package main;

public class Attributes {

public static class Person {

// ATTRIBUTE WITH PACKAGE ACCESS MODIFIER

/\* allows access from within the package \*/

String name;

// ATTRIBUTE WITH PRIVATE ACCESS MODIFIER

/\* restricts access to only within the class \*/

private int age;

// ATTRIBUTE WITH PROTECTED ACCESS MODIFIER

/\* allows access within the class and its subclasses \*/

protected int weight;

// ATTRIBUTE WITH PUBLIC ACCESS MODIFIER AND STATIC FIELD

/\* allows access from any classes, is the same for all objects due to

the static field \*/

public static String race = "human";

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

// GETTER FOR PRIVATE ATTRIBUTE

/\* needs getter (and setter) to access from outside the class \*/

public int getAge() {

return age;

}

public int getWeight() {

return weight;

}

public void setName(String name) {

this.name = name;

}

public void setAge(int age) {

this.age = age;

}

public void setWeight(int weight) {

this.weight = weight;

}

public void introduce() {

System.out.println("My name is " + name + ", and I am " + age + " years old.");

}

}

public static void main(String[] args) {

Person john = new Person("John", 25);

Person ida = new Person("Ida", 43);

john.introduce();

ida.introduce();

System.out.println(john.getName() + "is a " + john.race);

System.out.println(ida.getName() + "is a " + ida.race);

john.setWeight(97);

System.out.println(john.getName() + " weighs " + john.getWeight() + " kg");

System.out.println(ida.getName() + " is " + ida.getAge() + " years old");

}

}

Comparators

package main;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class Comparators {

public static void main(String[] args) {

List<Person> people = new ArrayList<>();

people.add(new Person("Alice", 20, 50000.50));

people.add(new Person("John", 24, 95000.00));

people.add(new Person("Marc", 17, 2000.50));

people.add(new Person("Cathy", 59, 105000.95));

people.add(new Person("Henry", 44, 25000.23));

people.add(new Person("Lucas", 36, 222000.22));

people.add(new Person("Yusuf", 9, 250.00));

reverseOrder(people);

sortAge(people);

sortMultipleCriteria(people);

}

// SORTING IN REVERSE ORDER

public static void reverseOrder(List<Person> people) {

Collections.sort(people, Comparator.comparing(Person::getName).reversed());

for (Person p : people) {

System.out.println(p.getName());

}

}

// SORTING BY AGE

public static void sortAge(List<Person> people) {

Collections.sort(people, Comparator.comparingInt(Person::getAge));

for (Person p : people) {

System.out.println(p.getName() + " - " + p.getAge());

}

}

// SORTING FIRST BY AGE, THEN BY SALARY

public static void sortMultipleCriteria(List<Person> people) {

Comparator<Person> comparator = Comparator.comparingInt(Person::getAge)

.thenComparingDouble(Person::getSalary);

Collections.sort(people,comparator);

for(Person p:people){

System.out.println(p.getName() + " - " + p.getAge() + " - " + p.getSalary());

}

}

static class Person {

private String name;

private int age;

private double salary;

public Person(String name, int age, double salary) {

this.name = name;

this.age = age;

this.salary = salary;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public double getSalary() {

return salary;

}

}

}

Conditional Branching

package main;

import java.util.Random;

public class ConditionalBranching {

public static void main(String[] args) {

Random rnd = new Random();

// IF

int x = rnd.nextInt(-1000, 1000);

System.out.println(x);

if (x > 0) {

System.out.println("x is positive");

}

// IF-ELSE

int age = rnd.nextInt(0, 100);

System.out.println(age);

if (age >= 18) {

System.out.println("You are an adult");

} else {

System.out.println("You are not an adult");

}

// IF-ELSE-IF-ELSE

int score = rnd.nextInt(0, 100);

System.out.println(score);

if (score >= 90) {

System.out.println("A");

} else if (score >= 80) {

System.out.println("B");

} else if (score >= 70) {

System.out.println("C");

} else {

System.out.println("D");

}

// SHORT IF

int y = rnd.nextInt(-1000, 1000);

System.out.println(y);

String result = (y > 0) ? "Positive" : "Negative";

System.out.println(result);

// SWITCH STATEMENT

int dayOfWeek = rnd.nextInt(1, 7);

System.out.println(dayOfWeek);

switch (dayOfWeek) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

default:

System.out.println("Invalid day");

}

// FALL-THROUGH-SWITCH

System.out.println(dayOfWeek);

switch (dayOfWeek) {

case 1:

case 2:

case 3:

case 4:

case 5:

System.out.println("Weekday");

break;

case 6:

case 7:

System.out.println("Weekend");

break;

default:

System.out.println("Invalid day");

}

// ENUMERATED SWITCH

enum Month {

JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,

JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER

}

Month month = Month.JULY;

switch (month) {

case DECEMBER:

case JANUARY:

case FEBRUARY:

System.out.println("Winter");

break;

case MARCH:

case APRIL:

case MAY:

System.out.println("Spring");

break;

case JUNE:

case JULY:

case AUGUST:

System.out.println("Summer");

break;

case SEPTEMBER:

case OCTOBER:

case NOVEMBER:

System.out.println("Autumn");

break;

default:

System.out.println("Invalid month");

}

// ARROW-FORM SWITCH

System.out.println(dayOfWeek);

String dayName = switch (dayOfWeek) {

case 1 -> "Monday";

case 2 -> "Tuesday";

case 3 -> "Wednesday";

case 4 -> "Thursday";

case 5 -> "Friday";

case 6 -> "Saturday";

case 7 -> "Sunday";

default -> "Invalid day";

};

System.out.println(dayName);

}

}

Constructors

package main;

public class Constructors {

public static class Person {

private String name;

private int age;

private String origin;

// DEFAULT CONSTRUCTOR

public Person() {

name = "";

}

// PARAMETERISED CONSTRUCTOR

public Person(String name, int age){

this.name = name;

this.age = age;

}

// OVERLOADED CONSTRUCTOR

public Person(String name, int age, String origin){

this.name = name;

this.age = age;

this.origin = origin;

}

public void introduce(){

System.out.println("I'm " + name + ", I'm " + age + " years old and I am " + origin);

}

}

public static void main(String[] args) {

Person ida = new Person();

Person john = new Person("John",99);

Person herbert = new Person("Herbert",76,"German");

ida.introduce();

john.introduce();

herbert.introduce();

}

}

Data Structures

package main;

import java.util.\*;

public class DataStructures {

public static void main(String[] args) {

// HASHMAP

/\* unique key-value pair that provides fast lookup and retrieval based on keys.

HashMap is useful when you need to store and retrieve data based on unique

keys, and when fast access to elements is required. \*/

HashMap<String, Integer> map = new HashMap<>();

HashMap<String, Integer> map2 = new HashMap<>();

/\* adding elements \*/

map.put("John", 25);

map.put("Alice", 13);

map2.put("Emily", 34);

map2.put("John", 25);

map2.put(null, null);

System.out.println(map);

System.out.println(map2);

/\* merge maps \*/

map.putAll(map2);

System.out.println(map);

/\* retrieving elements \*/

System.out.println(map.get("John"));

/\* removing elements \*/

map.remove("Alice");

map.remove(null);

System.out.println(map);

/\* checking existence \*/

System.out.println(map.containsKey("John") + ", " + map.containsValue(23));

/\* Size \*/

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

// HASHTABLE

/\* unique key-value pair that provides fast lookup and retrieval based on keys. HashMap is useful when you need to store and retrieve data based on unique keys, and when fast access to elements is required. HashTable is thread-safe, but has lower performance. It also doesn't allow null keys or values. \*/

Hashtable<String, Integer> metals = new Hashtable<>();

/\* adding elements \*/

metals.put("Gold", 18);

metals.put("Platinum", 24);

metals.put("Copper", 14);

metals.put("Copper", 14);

try {

metals.put(null, 23);

} catch (NullPointerException n) {

System.out.println("null not allowed");

}

System.out.println(metals);

/\* retrieving elements \*/

System.out.println(metals.get("Platinum"));

/\* removing elements \*/

metals.remove("Copper");

System.out.println(metals);

/\* checking existence \*/

System.out.println(metals.containsKey("Copper") + ", " + metals.containsValue(18));

/\* Size \*/

System.out.println(metals.size());

// HASHSET

/\* unordered collection of unique elements, does not allow duplicates.

HashSet is useful when you want to store a collection of unique elements

and don't require a specific order. \*/

HashSet<String> set = new HashSet<>();

/\* adding elements \*/

set.add("Apple");

set.add("Banana");

set.add("Orange");

set.add("Apple");

System.out.println(set);

/\* iterate \*/

Iterator<String> hSIterator = set.iterator();

while (hSIterator.hasNext()) {

System.out.print(hSIterator.next() + ", ");

}

System.out.println();

/\* removing elements \*/

set.remove("Banana");

System.out.println(set);

/\* checking existence \*/

System.out.println(set.contains("Apple"));

/\* Size \*/

System.out.println(set.size());

// ARRAYLIST

/\* dynamic array implementation that provides fast random access and

dynamic resizing. ArrayList is useful when you need a resizable

array-like structure that provides fast access to elements.\*/

ArrayList<String> list = new ArrayList<>();

/\* adding elements \*/

list.add("red");

list.add("blue");

list.add("yellow");

list.add("red");

System.out.println(list);

/\* retrieving elements \*/

System.out.println(list.get(2));

/\* removing elements \*/

list.remove(2);

System.out.println(list.get(2));

System.out.println(list);

/\* Size \*/

System.out.println(list.size());

// ARRAYDEQUE

/\* ArrayDeque is a double-ended queue that allows efficient

insertion and removal at both ends. ArrayDeque is useful when

you need to implement a queue or stack data structure with

efficient insertion and removal operations at both ends.\*/

ArrayDeque<String> deque = new ArrayDeque<>();

/\* adding elements \*/

deque.addFirst("one");

deque.addLast("Two");

deque.addLast("three");

deque.addFirst("three");

System.out.println(deque);

/\* retrieving elements \*/

System.out.println(deque.getFirst() + ", " + deque.getLast());

/\* removing elements \*/

deque.removeFirst();

System.out.println(deque);

/\* Size \*/

System.out.println(deque.size());

// LINKEDLIST

/\* doubly linked list implementation that provides efficient insertion

and removal operations at both ends, as well as fast access to elements

using iterators. LinkedList is useful when you need to frequently insert

and remove elements at the beginning or end of the list, or when you

need to iterate over the elements.\*/

LinkedList<String> linkedList = new LinkedList<>();

/\* adding elements \*/

linkedList.add("Hispanic");

linkedList.add("White");

linkedList.add("Black");

linkedList.add("White");

linkedList.addFirst("Brown");

System.out.println(linkedList);

/\* checking occurrence \*/

System.out.println(linkedList.indexOf("Black"));

System.out.println(linkedList.lastIndexOf("White"));

System.out.println(linkedList.contains("Indian"));

/\* retrieving elements \*/

System.out.println(linkedList.getFirst() + ", " + linkedList.getLast());

System.out.println(linkedList.get(3));

/\* iterating over the elements \*/

Iterator<String> iterator = linkedList.iterator();

while (iterator.hasNext()) {

System.out.print(iterator.next() + ", ");

}

System.out.println();

/\* removing elements \*/

linkedList.removeLast();

System.out.println(linkedList);

/\* Size \*/

System.out.println(linkedList.size());

// STACK

/\* Stack is a specialized data structure that follows the

Last-In-First-Out (LIFO) principle. It provides push and pop

operations. Stack is useful when you need to implement a LIFO

behavior, such as in algorithms that require backtracking or

undo functionality.\*/

Stack<String> stack = new Stack<>();

/\* pushing elements \*/

stack.push("Cube");

stack.push("Sphere");

stack.push("Pyramid");

stack.push("Cube");

System.out.println(stack);

/\* popping elements \*/

System.out.println(stack.pop());

System.out.println(stack);

/\* checking existence \*/

System.out.println(stack.isEmpty());

/\* Size \*/

System.out.println(stack.size());

// TREESET

/\* TreeSet is an ordered set implementation based on a self-balancing

binary search tree (Red-Black Tree). It maintains elements in sorted

order. TreeSet is useful when you need to maintain elements in a sorted

order and efficiently perform operations like retrieving the first or

last element. \*/

TreeSet<String> treeSet = new TreeSet<>();

/\* adding elements \*/

treeSet.add("Fir");

treeSet.add("Birch");

treeSet.add("Poplar");

treeSet.add("Fir");

System.out.println(treeSet);

/\* retrieving elements \*/

System.out.println(treeSet.first() + ", " + treeSet.last());

/\* Removing elements \*/

treeSet.remove("Fir");

System.out.println(treeSet);

/\* Size \*/

System.out.println(treeSet.size());

// PRIORITYQUEUE

/\* orders its elements based on their natural order or a custom

comparator provided during its creation \*/

PriorityQueue<String> priorityQueue = new PriorityQueue<>();

/\* adding elements \*/

priorityQueue.offer("Honda"); //if the queue is full, no exception is thrown

priorityQueue.add("Suzuki"); // if the queue is full, throws exception

priorityQueue.addAll(Arrays.asList("Yamaha", "Ducati", "BMW"));

System.out.println(priorityQueue);

/\* retrieving the head element \*/

System.out.println(priorityQueue.peek());

/\* removing the head element \*/

System.out.println(priorityQueue.poll());

System.out.println(priorityQueue);

// CUSTOM PRIORITYQUEUE WITH COMPARABLE INTERFACE

class Person implements Comparable<Person> {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public Integer getAge() {

return age;

}

@Override

public int compareTo(Person other) {

return Integer.compare(this.age, other.age);

}

}

PriorityQueue<Person> pq = new PriorityQueue<>();

pq.addAll(Arrays.asList(new Person("Alice", 15), new Person("Bob", 35), new Person("Yuki", 19), new Person("Sarah", 48)));

while (!pq.isEmpty()) {

Person person = pq.poll();

System.out.println(person.getName() + ", " + person.getAge());

}

}

}

Enums

package main;

import java.util.EnumSet;

public class Enums {

// DECLARATION

/\* declared as public, final and static \*/

enum Day {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

// DECLARATION WITH A CONSTRUCTOR

enum GreekLetters {

ALPHA("a"), BETA("b"), GAMMA("c");

private String latin;

GreekLetters(String latin) {

this.latin = latin;

}

public String getLatin() {

return latin;

}

}

// DECLARATION WITH MULTIPLE VALUES

enum Size {

SMALL("S", 32), MEDIUM("M", 36), LARGE("L", 40);

private String code;

private int measurement;

Size(String code, int measurement) {

this.code = code;

this.measurement = measurement;

}

public String getCode() {

return code;

}

public int getMeasurement() {

return measurement;

}

}

enum AllMonths {

JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER

}

public static void main(String[] args) {

// ACCESSING ENUM CONSTANTS

Day today = Day.MONDAY;

System.out.println(today);

// ITERATE THROUGH ENUM

for (GreekLetters letters : GreekLetters.values()) {

System.out.print(letters + ":" + letters.getLatin() + ", ");

}

System.out.println();

System.out.println(Size.MEDIUM.getMeasurement());

// FIND ORDINAL OF ENUMS

Day tomorrow = Day.SATURDAY;

System.out.println(tomorrow.ordinal());

// GET AN ARRAY OF ALL CONSTANTS

Day[] days = Day.values();

for (Day d : days) {

System.out.print(d + " ");

}

System.out.println();

// CREATE AN ENUMSET AS ENUM SUBCLASS

EnumSet<AllMonths> firstSemester = EnumSet.range(

AllMonths.JANUARY, AllMonths.JUNE);

System.out.println(firstSemester);

// ADD CONSTANTS TO THE ENUMSET

firstSemester.add(AllMonths.JULY);

System.out.println(firstSemester);

// REMOVE CONSTANTS FROM THE ENUMSET

firstSemester.remove(AllMonths.JULY);

System.out.println(firstSemester);

}

}

Error Handling

Package main;

import java.io.FileReader;

import java.io.IOException;

import java.util.Scanner;

public class ErrorHandling {

public static void main(String[] args) {

// CATCHING AND HANDLING SPECIFIC EXCEPTIONS

try {

int result = 10 / 0;

System.out.println(result);

} catch (ArithmeticException e) {

System.out.println("Dividing by 0 not possible");

}

// CATCHING MULTIPLE EXCEPTIONS

try {

int[] result2 = {2, 5, 3, 5, 9, 1};

System.out.println(result2[11]);

} catch (ArithmeticException e) {

System.out.println("Dividing by 0 not possible");

} catch (Exception f) {

System.out.println("not possible");

}

try {

div(10, 0);

div(10, 1);

} catch (Exception e) {

}

// USING A FINALLY BLOCK

FileReader fileReader = null;

try {

fileReader = new FileReader("file.txt");

} catch (IOException e) {

System.out.println("An error occurred while reading the file: " + e.getMessage());

} finally {

if (fileReader != null) {

try {

fileReader.close();

} catch (IOException e) {

System.out.println("Cannot close the file " + e.getMessage());

}

}

}

try{

String input = null;

processInput(input);

} catch (invalidInputException e){

System.out.println(e.getMessage());

}

}

// THROWING EXCEPTIONS

static int div(int dividend, int divisor) throws ArithmeticException {

if (divisor == 0) {

throw new ArithmeticException("Cannot divide by zero.");

}

return dividend / divisor;

}

// CREATING CUSTOM EXCEPTIONS

static class invalidInputException extends Exception {

public invalidInputException(String message) {

super(message);

}

}

public static void processInput(String input) throws invalidInputException{

if (input == null || input.isEmpty()){

throw new invalidInputException("Input is invalid or empty");

}

}

}

File Handling

package main;

import java.io.\*;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.StandardCopyOption;

import java.util.Random;

public class FileHandling {

public static void main(String[] args) {

Random random = new Random();

// GET THE PATH OF THE CURRENT DIRECTORY

File currentDir = new File(".");

System.out.println(currentDir.getAbsolutePath());

// MODIFY THE PATH OF THE CURRENT DIRECTORY

File src = new File(currentDir + "/src/");

System.out.println(src.getAbsolutePath());

// FINDING THE CURRENT USER'S DIRECTORY

File user = new File(System.getProperty("user.dir"));

System.out.println(user.getAbsolutePath());

// SYSTEM.GETPROPERTY COMMANDS

/\* java.version Java Runtime Environment version

java.class.path location of Java class files and libraries

java.library.path location of native libraries

os.name Operating system name

os.arch Operating system architecture

os.version Operating system version

file.separator File separator character

path.separator Path separator character

line.separator Line separator character

user.name User account name

user.home User's home directory

user.dir Current working directory

user.language User's preferred language

user.timezone User's preferred time zone

sun.desktop User's desktop environment \*/

// CREATING A FOLDER

File folder = new File(src + "/files/");

if (!folder.exists()) {

boolean created = folder.mkdir();

if (created) {

System.out.println("Folder created: " + folder.getAbsolutePath());

} else {

System.out.println("Folder not created");

}

} else {

System.out.println("Folder already exists");

}

// CREATING A FILE

File file = new File(src + "/text.txt/");

try {

boolean created = file.createNewFile();

if (created) {

System.out.println("File created: " + file.getAbsolutePath());

} else {

System.out.println("File already exists");

}

} catch (IOException e) {

e.printStackTrace();

}

// WRITING INTO THE FILE

try (BufferedWriter writer = new BufferedWriter(new FileWriter(file))) {

for (int i = 0; i < 100; i++) {

int rnd = random.nextInt(64, 129);

char chr = (char) rnd;

writer.write(chr);

}

System.out.println("Data written to file.");

} catch (IOException e) {

e.printStackTrace();

}

// READING FROM THE FILE

try (BufferedReader reader = new BufferedReader(new FileReader(file))) {

String line;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException e) {

e.printStackTrace();

}

// MOVING A FILE

try {

Path source = file.toPath();

Path targetFolder = folder.toPath().resolve(file.getName());

Files.move(source, targetFolder, StandardCopyOption.REPLACE\_EXISTING);

System.out.println("File moved to: " + targetFolder.toAbsolutePath());

} catch (IOException e) {

e.printStackTrace();

}

// COPYING A FILE

try {

Path source = new File("." + "/src/files/text.txt").toPath();

Path target = new File("." + "/src/files/text\_copy.txt").toPath();

Files.copy(source, target, StandardCopyOption.REPLACE\_EXISTING);

System.out.println("Files copied to: " + target.toAbsolutePath());

} catch (IOException e) {

e.printStackTrace();

}

// DELETING A FILE

try {

Path target = new File("." + "/src/files/text.txt").toPath();

Files.deleteIfExists(target);

System.out.println("File deleted: " + target.toAbsolutePath());

} catch (IOException e) {

e.printStackTrace();

}

// LISTING CONTENTS OF A FOLDER

File fileA = new File("." + "/src/files/fileA.txt");

File fileB = new File("." + "/src/files/fileB.txt");

try {

fileB.createNewFile();

fileA.createNewFile();

} catch (IOException e) {

e.printStackTrace();

}

File[] files = folder.listFiles();

if (files != null) {

for (File f : files) {

System.out.println(f.getName());

}

}

// DELETING CONTENTS OF A FOLDER

Path deleteContents = new File("." + "/src/files").toPath();

if (files != null) {

for (File f : files) {

if (!(f.getName().contains("text"))) {

f.delete();

}

}

}

}

}

Generics

package main;

import java.util.ArrayList;

public class Generics {

// GENERIC CLASS

/\* can create a Pocket object with an ArrayList of any type of contents \*/

static class Pocket<T> {

private ArrayList<T> contents;

public void setContents(ArrayList<T> contents) {

this.contents = contents;

}

public ArrayList<T> getContents() {

return contents;

}

// GENERIC METHOD WITH BOUNDED TYPE PARAMETER

/\* can use any type of ArrayList, in this case it is bounded to any

single-type child class of Number, e.g. Integer or Double \*/

public <T extends Number> void calculateBills(ArrayList<T> pocket) {

double sum = 0;

for (T money : pocket) {

sum += money.doubleValue();

}

System.out.println(sum);

}

// GENERIC METHOD WITH BOUNDED WILDCARD

/\* can use any type of ArrayList, can work with ArrayLists that

contain elements from diverse classes, as long as they are

children of Number, e.g. Integer, Double, Float \*/

public void calculateMoney(ArrayList<? extends Number> pocket) {

double sum = 0;

for (Number money : pocket) {

sum += money.doubleValue();

}

System.out.println(sum);

}

}

public static void main(String[] args) {

// GENERIC OBJECT

/\* can only be filled with strings \*/

Pocket<String> cardPocket = new Pocket<>();

ArrayList<String> cards = new ArrayList<>();

cards.add("Visa");

cards.add("ID");

cardPocket.setContents(cards);

Pocket<Integer> billPocket = new Pocket<>();

ArrayList<Integer> bills = new ArrayList<>();

bills.add(10);

bills.add(100);

bills.add(50);

billPocket.setContents(bills);

billPocket.calculateBills(bills);

// GENERIC WILDCARD OBJECT

/\* can be filled with elements from any class \*/

Pocket<?> coinPocket = new Pocket<>();

/\* can be filled with elements from any child class of Number \*/

ArrayList<Number> coins = new ArrayList<>();

coins.add(0.5);

coins.add(2);

coins.add(0.1);

coins.add(0.2);

coins.add(0.2);

coins.add(5);

coinPocket.calculateMoney(coins);

}

}

Inheritance

package main;

public class Inheritance {

// PARENT CLASS / SUPERCLASS

static class Animal {

protected String name;

public Animal(String name) {

this.name = name;

}

public void speak() {

System.out.println("Animal speaks");

}

}

// SUBCLASS / CHILD

static class Cat extends Animal {

private String breed;

/\* Inherited constructor from parent, but overwritten \*/

public Cat(String name, String breed) {

super(name);

this.breed = breed;

}

/\* overridden method from parent \*/

@Override

public void speak() {

System.out.println("Cat meows");

}

/\* added method \*/

public void purr() {

System.out.println("Cat purrs");

}

}

public static void main(String[] args) {

Animal animal = new Animal("Bello");

animal.speak();

Cat cat = new Cat("Whiskers", "Persian");

cat.speak();

cat.purr();

}

}

Interfaces

package main;

public class Interfaces {

public static void main(String[] args) {

Apple apple = new Apple();

Tequila tequila = new Tequila();

// STATIC METHOD THAT CAN ONLY BE CALLED BY ITS INTERFACE

Ripe.isRipe(true);

// METHODS THAT CAN BE CALLED BY THEIR OBJECTS

apple.isDisgusting(true, false);

tequila.isDisgusting(true, false);

}

// CLASS IMPLEMENTING DISGUSTING AND RIPE INTERFACE

public static class Apple implements Disgusting, Ripe {

boolean worm, fungus;

public Apple() {

}

/\* Method has to be implemented because the Disgusting interface inherits

the Worm interface \*/

@Override

public boolean hasWorm(boolean worm) {

return worm ? true : false;

}

/\* isDisgusting does not need to be implemented because it is a default method \*/

/\* isRipe does not need to be implemented because it is a static method \*/

/\* hasFungus does not need to be implemented because it is a static method \*/

}

// CLASS IMPLEMENTING DISGUSTING INTERFACE

public static class Tequila implements Disgusting {

boolean worm, fungus;

public Tequila() {

}

/\* Method has to be implemented because the Disgusting interface inherits

the Worm interface \*/

@Override

public boolean hasWorm(boolean worm) {

return worm ? true : false;

}

/\* isDisgusting can be implemented because it is a default method and can be overridden \*/

@Override

public void isDisgusting(boolean worm, boolean fungus) {

if (worm && fungus) {

System.out.println("disgusting");

} else if (fungus) {

System.out.println("not great");

} else {

System.out.println("yummy");

}

}

/\* isRipe does not need to be implemented because it is a static method \*/

/\* hasFungus does not need to be implemented because it is a static method \*/

}

// INTERFACE WITH METHOD

/\* has to be implemented in every class that inherits interface \*/

public interface Worm {

public boolean hasWorm(boolean worm);

}

// INTERFACE WITH STATIC METHOD

/\* does not need to be explicitly implemented, is the same for all classes

that inherit it, can only be called by the interface name \*/

public interface Fungus {

public static boolean hasFungus(boolean fungus) {

return fungus ? true : false;

}

}

// INTERFACE INHERITING FROM WORD AND FUNGUS WITH DEFAULT METHOD

/\* does not need to be explicitly implemented, is the same for all classes

that inherit it unless it is overridden \*/

public interface Disgusting extends Worm, Fungus {

default void isDisgusting(boolean worm, boolean fungus) {

if (worm && fungus) {

System.out.println("disgusting");

} else if (worm || fungus) {

System.out.println("not great");

} else {

System.out.println("yummy");

}

}

}

// INTERFACE WITH STATIC METHOD

/\* does not need to be explicitly implemented, is the same for all classes

that inherit it, can only be called by the interface name \*/

public interface Ripe {

public static void isRipe(boolean ripe) {

System.out.println(ripe ? "is ripe" : "is not ripe");

}

}

}

Lambdas

package main;

import java.lang.reflect.Array;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.StringJoiner;

import java.util.function.\*;

import static java.lang.Math.\*;

public class Lambdas {

// LAMBDA WITH SINGLE PARAMETER

/\* Syntax: Function<input,output> || T -> R \*/

static Function<Double, Double> square = x -> pow(x.doubleValue(), 2);

// LAMBDA WITH MULTIPLE PARAMETERS

/\* Syntax: BiFunction<input,input,output> || T,R -> S \*/

static BiFunction<Integer, Integer, Integer> sum = (a, b) -> {

return a + b;

};

// PREDICATE LAMBDA

/\* boolean-valued function of one argument || T -> boolean \*/

static Predicate<Double> isEven = x -> x % 2 == 0;

static Predicate<Double> isGT4 = e -> e > 4;

// CONSUMER LAMBDA

/\* represents an operation that takes a single argument and returns no result || T -> void \*/

static Consumer<String> printer = message -> System.out.println(message);

// FUNCTION LAMBDA

/\* takes one argument and produces a result || T -> R \*/

static Function<Integer, String> stringify = x -> String.valueOf(x);

// SUPPLIER LAMBDA

/\* represents a supplier of results, no input needed || () -> T \*/

static Supplier<Double> randomValue = () -> Math.random() \* 100;

// LAMBDA WITH METHOD REFERENCE

/\* reference to a static method \*/

static Function<Integer, Double> squareRoot = Math::sqrt;

// ARBITRARTY OBJECT METHOD REFERENCE

static Function<String, Integer> lengthGetter = String::length;

// TYPE INFERENCE

static Function<Integer, String> binStringConverter = (n) -> Integer.toBinaryString(n);

public static void main(String[] args) {

System.out.println(square.apply(3.14));

System.out.println(sum.apply(5, 9));

System.out.println(isEven.test(6.0));

System.out.println(isGT4.test(5.5));

System.out.println(isGT4.and(isEven).test(6.0));

printer.accept("Hi");

System.out.println(stringify.apply(45));

System.out.println(randomValue.get());

System.out.println(squareRoot.apply(559));

StringJoiner joiner = new StringJoiner(", ");

StringJoiner joiner2 = new StringJoiner("; ");

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// INSTANCE METHOD REFERENCE

names.forEach(joiner::add);

/\* does the same as the forEach loop below \*/

for (String s : names) {

joiner2.add(s);

}

System.out.println(joiner.toString());

System.out.println(joiner2.toString());

// CONSTRUCTOR METHOD REFERENCE

Supplier<ArrayList<String>> listSupplier = ArrayList::new;

/\* is equivalent to the first example \*/

Supplier<ArrayList<String>> listSupplier1 = () -> new ArrayList<>();

/\* saves the explicit definition of an anonymous inner class that implements

the Supplier<ArrayList<String>> interface and overrides the get() method, where

you have to instantiate a new object using the constructor. \*/

Supplier<ArrayList<String>> listSupplier2 = new Supplier<ArrayList<String>>() {

@Override

public ArrayList<String> get() {

return new ArrayList<String>();

}

};

ArrayList<String> list = listSupplier.get();

ArrayList<String> list1 = listSupplier2.get();

ArrayList<String> list2 = listSupplier2.get();

System.out.println(list);

System.out.println(list1);

System.out.println(list2);

System.out.println(lengthGetter.apply("This is a String with a length"));

String message = "Run, Forrest!";

System.out.println(binStringConverter.apply(42));

// VARIABLE CAPTURE

Runnable runnable = () -> System.out.println(message);

runnable.run();

}

}

Loops

package main;

public class Loops {

public static void main(String[] args) {

// FOR-LOOP

/\* Is used when you know the number of iterations in advance. \*/

for (int a = 1; a <= 5; a++) {

System.out.print(a + " ");

}

// WHILE-LOOP

/\* Is used when the number of iterations is not known in advance,

and the loop continues as long as a certain condition is true. \*/

int b = 1;

while (b <= 5) {

System.out.print(b + " ");

b++;

}

// DO-WHILE-LOOP

/\* Is similar to the while loop, but it ensures that the loop body

executes at least once before checking the condition. \*/

int c = 1;

do {

System.out.print(c + " ");

c++;

} while (c <= 5);

// ENHANCED FOR-LOOP or FOR-EACH-LOOP

/\* Is used to iterate over arrays or collections. \*/

String[] names = {"John", "Mary", "Alex"};

for (String name : names) {

System.out.print(name + " ");

}

// NESTED LOOPS

/\* You can have loops within loops, known as nested loops, to perform

iterative operations on multiple dimensions or nested structures. \*/

for (int d = 1; d <= 5; d++) {

for (int e = 1; e <= d; e++) {

System.out.print("\* ");

}

}

}

}

Math Library

package main;

public class MathLibrary {

public static void main(String[] args) {

// ABSOLUTE VALUE

System.out.println(Math.abs(-50)); //Returns the absolute value

// ARC COSINE

System.out.println(Math.acos(1)); //Returns the arc cosine of a value

// ARC SINE

System.out.println(Math.asin(1)); //Returns the arc sine of a value

// ARC TANGENT

System.out.println(Math.atan(1)); //Returns the arc tangent of a value

// CUBE ROOT

System.out.println(Math.cbrt(27)); //Returns the cube root of a double value

// COSINE

System.out.println(Math.cos(Math.PI)); //Returns the trigonometric cosine of an angle

// E^X

System.out.println(Math.exp(2)); //Returns Euler 's number e raised to the power of a double value

// E^-1

System.out.println(Math.expm1(2)); //Returns ex -1

// LOGARITHM

System.out.println(Math.log(10)); //Returns the natural logarithm of a value

// LOG 10

System.out.println(Math.log10(10)); //Returns the base 10 logarithm of a value

// MAXIMUM

System.out.println(Math.max(5,10)); //Returns the greater of two values

// MINIMUM

System.out.println(Math.min(5,10)); //Returns the smaller of two values.

// EXPONENTIATION

System.out.println(Math.pow(5,2)); //Returns the value of the first argument raised to the power of the second argument

// RANDOM

System.out.println(Math.random()); //Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0

// ROUND INTEGER

System.out.println(Math.rint(5.27)); //Returns the double value that is closest in value to the argument and is equal to a mathematical integer

// ROUND

System.out.println(Math.round(5.225)); //Returns the closest int to the argument, with ties rounding to positive infinity

// SINE

System.out.println(Math.sin(Math.PI)); //Returns the trigonometric sine of an angle

// SQUARE ROOT

System.out.println(Math.sqrt(81)); //Returns the correctly rounded positive square root of a double value

// TANGENT

System.out.println(Math.tan(Math.PI)); //Returns the trigonometric tangent of an angle

// RADIAN TO DEGREE

System.out.println(Math.toDegrees(1.45)); //Converts an angle measured in radians to an approximately equivalent angle measured in degrees

// DEGREE TO RADIAN

System.out.println(Math.toRadians(90)); //Converts an angle measured in radians to an approximately equivalent angle measured in degrees.

}

}

Methods

package main;

public class Methods {

public static void main(String[] args) {

Methods methods = new Methods();

double[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};

methods.start();

methods.printSum(5, 10);

int sum = methods.calculateSum(55, 10);

System.out.println(sum);

methods.getGreeting();

double f = methods.calculateAverage(numbers);

System.out.println(f);

int a = methods.factorial(10);

System.out.println(a);

int b = methods.findMax(345, 645, 5, 123, 2, 535, 234, 23, 4);

System.out.println(b);

System.out.println(PiExponentiation(14));

}

// METHOD WITHOUT PARAMETERS AND RETURN VALUE

void start() {

System.out.println("Let's calculate:");

}

// METHOD WITH PARAMETERS BUT NO RETURN VALUE

void printSum(int a, int b) {

int sum = a + b;

System.out.println("Sum: " + sum);

}

// METHOD WITH PARAMETERS AND RETURN VALUE

int calculateSum(int a, int b) {

int sum = a + b;

return sum;

}

// METHOD WITH RETURN VALUE BUT NO PARAMETERS

private String getGreeting() {

return "So far, so good";

}

// METHOD WITH MULTIPLE PARAMETERS AND RETURN VALUE

double calculateAverage(double[] numbers) {

double sum = 0;

for (double num : numbers) {

sum += num;

}

double average = sum / numbers.length;

return average;

}

// RECURSIVE METHOD

int factorial(int n) {

if (n == 0 || n == 1) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

// METHOD WITH MULTIPLE PARAMETERS AND RETURN VALUE

int findMax(int... numbers) {

int max = Integer.MIN\_VALUE;

for (int num : numbers) {

if (num > max) {

max = num;

}

}

return max;

}

// STATIC RECURSIVE METHOD

/\* Accessing Static main.Methods: Since static methods are associated with the

class itself, they can be accessed directly using the class name, followed

by the method name. No Instance Dependency: Static methods do not depend on

any specific instance of the class. They can be called even if no objects of

the class are created. \*/

public static int PiExponentiation(int n) {

if (n == 1) {

return 1;

} else {

return (int) Math.pow(Math.PI, PiExponentiation(n - 1));

}

}

}

Operators

package main;

import java.sql.SQLOutput;

public class Operators {

public static void main(String[] args) {

int a = 1, b = 2, c = 3, d = 4;

// ARITHMETIC OPERATORS

System.out.println(a + b); //Addition

System.out.println(d - c); //Subtraction

System.out.println(b \* c); //Multiplication

System.out.println(d / b); //Division

System.out.println(d % c); //Modulus

// ASSIGNMENT OPERATORS

System.out.println(b);

b += 5; // x = x + 5

System.out.println(b);

b -= 3; // x = x - 3

System.out.println(b);

b \*= 2; // x = x \* 2

System.out.println(b);

b /= 4; // x = x / 4

System.out.println(b);

b %= 3; // x = x % 3

System.out.println(b);

// COMPARISON OPERATORS

System.out.println((a == b)); // Equality: false

System.out.println((a != b)); // Inequality: true

System.out.println((a > b)); // Greater than: false

System.out.println((a < b)); // Less than: true

System.out.println((a >= b)); // Greater than or equal to: false

System.out.println((a <= b)); // Less than or equal to: true

// INCREMENT AND DECREMENT

System.out.println(c);

System.out.println(d);

c++;

d--;

System.out.println(c);

System.out.println(d);

// TENARY OPERATOR

int max = (a > b) ? a : b;

System.out.println(max);

}

}

Random Values

package main;

import java.util.Random;

public class RandomValues {

public static void main(String[] args) {

// INITIALIZE

Random random = new Random();

// RANDOM INTEGER

System.out.println(random.nextInt());

// RANDOM POSITIVE INTEGER BELOW MAX

/\* Syntax: nextInt(Max) \*/

System.out.println(random.nextInt(100));

// RANDOM INTEGER BETWEEN MIN AND MAX

/\* Syntax: nextInt(Min,Max) \*/

System.out.println(random.nextInt(-1000, 1000));

// RANDOM DOUBLE

System.out.println(random.nextDouble());

// RANDOM BOOLEAN

System.out.println(random.nextBoolean());

}

}

Regular Expressions

package main;

import org.w3c.dom.Text;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegEx {

public static void main(String[] args) {

// PATTERN MATCHING

/\* used to match patterns in text. They allow you to search,

validate, and manipulate strings based on specific patterns \*/

String pattern = "\\d{4}-\\d{2}-\\d{2}";

String text = "Today's date is 2022-09-15";

boolean isMatch = text.matches(".\*" + pattern + ".\*");

System.out.println(isMatch);

// . (DOT)

/\* Matches any single character. \*/

pattern = "c.t";

text = "cat, cot, cut";

Pattern regex = Pattern.compile(pattern);

Matcher matcher = regex.matcher(text);

while (matcher.find()) {

System.out.println(matcher.group());

}

// \* (ASTERISK)

/\* Matches zero or more occurrences of the preceding character or group. \*/

pattern = "ab\*c";

text = "ac, abc, abbc, abbbc";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// + (PLUS)

/\* Matches one or more occurrences of the preceding character or group. \*/

pattern = "ab+c";

text = "ac, abc, abbc, abbbc";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// ? (QUESTION MARK)

/\* Matches zero or one occurrence of the preceding character or group. \*/

pattern = "colou?r";

text = "color, colour";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// \ (BACKSLASH)

/\* Escapes special characters or indicates character classes. \*/

pattern = "\\[abc\\]";

text = "[abc] is a character class";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// [ ] (SQUARE BRACKETS)

/\* Defines a character class. Matches any character within the brackets. \*/

pattern = "[aeiou]";

text = "Hello World";

regex = Pattern.compile(pattern, Pattern.CASE\_INSENSITIVE);

matcher = regex.matcher(text);

while (matcher.find()) {

System.out.println(matcher.group());

}

// ^ (CARET)

/\* Matches the beginning of a line or string. \*/

pattern = "^Start";

text = "Start of the line\nAnother line starts with Start";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// $ (DOLLAR)

/\* Matches the end of a line or string. \*/

pattern = "end$";

text = "This is the end\nAnother line ends with end";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// \d

/\* Matches any digit (0-9) \*/

pattern = "\\d+";

text = "12345";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// \w

/\* Matches any word character (a-z, A-Z, 0-9 and underscore \*/

pattern = "\\w+";

text = "hello\_world";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// \s

/\* Matches any whitespace character \*/

pattern = "\\s+";

text = "Hello World";

regex = Pattern.compile(pattern);

matcher = regex.matcher(text);

while (matcher.find()) {

System.out.println(matcher.group());

}

// \D

/\* Matches any non-digit character \*/

pattern = "\\D+";

text = "Hello123";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// \W

/\* Matches any non-word character \*/

pattern = "\\W+";

text = "Hello@World";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// \S

/\* Matches any non-whitespace character \*/

pattern = "\\S+";

text = "Hello\tWorld";

regex = Pattern.compile(pattern);

matcher = regex.matcher(text);

while (matcher.find()) {

System.out.println(matcher.group());

}

// {n}

/\* Matches exactly n occurrences of the preceding character or group. \*/

pattern = "a{3}";

text = "aaa";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// {n,}

/\* Matches at least n occurrences of the preceding character or group. \*/

pattern = "a{3,}";

text = "aaaa";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// {n,m}

/\* Matches between n and m occurrences of the preceding character or group. \*/

pattern = "a{2,4}";

text = "aaa";

isMatch = text.matches(pattern);

System.out.println(isMatch);

// \b

/\* Matches a word boundary. \*/

pattern = "\\bworld\\b";

text = "Hello world! Welcome to the world.";

regex = Pattern.compile(pattern, Pattern.CASE\_INSENSITIVE);

matcher = regex.matcher(text);

while (matcher.find()) {

System.out.println(matcher.group());

}

// GROUPS AND CAPTURING

/\* Groups allow you to capture parts of a matched string. \*/

pattern = "(\\d+)-(\\w+)";

text = "123-abc";

regex = Pattern.compile(pattern);

matcher = regex.matcher(text);

if (matcher.matches()) {

String wholeMatch = matcher.group(0);

String group1 = matcher.group(1);

String group2 = matcher.group(2);

System.out.println(wholeMatch);

System.out.println(group1);

System.out.println(group2);

}

// EXTRACTING EMAIL ADDRESSES FROM TEXT

pattern = "\\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}\\b";

text = "Contact us at info@example.com or support@example.com";

Pattern emailPattern = Pattern.compile(pattern);

matcher = emailPattern.matcher(text);

while (matcher.find()) {

String email = matcher.group();

System.out.println(email);

}

// VALIDATING A PHONE NUMBER

pattern = "\\d{3} \\d{3} \\d{2} \\d{2}";

String phoneNumber = "079 960 51 42";

boolean isValid = phoneNumber.matches(pattern);

System.out.println(isValid);

// SPLITTING TEXT INTO WORDS

pattern = "\\W+"; //Non-word characters as delimiter

text = "Hello, world! How are you today?";

String[] words = text.split(pattern);

for (String word : words) {

System.out.println(word);

}

// REPLACING TEXT

pattern = "\\bapple\\b";

text = "I have an apple and a banana. I love apples.";

String replacedText = text.replaceAll(pattern, "orange");

System.out.println(replacedText);

// EXTRACTING DOMAIN NAME FROM URLs

pattern = "https?://([\\w.-]+)";

String url = "https://www.example.com";

regex = Pattern.compile(pattern);

matcher = regex.matcher(url);

if (matcher.find()) {

String domain = matcher.group(1);

System.out.println(domain);

}

// REMOVING HTML TAGS FROM TEXT

pattern = "<[^>]+>";

text = "<p>Hello, <b>world!</b></p>";

String strippedText = text.replaceAll(pattern, "");

System.out.println(strippedText);

// EXTRACTING HASHTAGS FROM SOCIAL MEDIA POSTS

pattern = "#(\\w+)";

String post = "Having a great time at #vacation. #sun #beach";

regex = Pattern.compile(pattern);

matcher = regex.matcher(post);

while (matcher.find()) {

String hashtag = matcher.group(1);

System.out.println(hashtag);

}

}

}

Streams

package main;

import java.util.Arrays;

import java.util.FormatterClosedException;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class Streams {

public static void main(String[] args) {

List<String> languages1 = Arrays.asList("German", "Hindi", "Mandarin");

String[] languages2 = {"English", "Spanish"};

// CREATING A STREAM

/\* from a collection \*/

Stream<String> streamFromCollection = languages1.stream();

/\* from an array \*/

Stream<String> streamFromArray = Arrays.stream(languages2);

/\* from specified elements \*/

Stream<String> streamOfElements = Stream.of("French", "Italian", "Russian");

/\* creating an infinite stream \*/

Stream<Integer> infiniteStream = Stream.iterate(1, n -> n + 1);

List<Integer> numbers = infiniteStream.limit(25).collect(Collectors.toList());

System.out.println(numbers);

numbers.addAll(Arrays.asList(2, 5, 8, 66, 5, 9, 4, 8, 1, 2, 3, 5, 9));

// MERGING STREAMS

Stream<String> lng = Stream.concat(streamFromArray, Stream.concat(streamOfElements, streamFromCollection));

List<String> languages = lng.collect(Collectors.toList());

System.out.println(languages);

// FILTER EVEN NUMBERS

numbers.stream()

.filter(n -> n % 2 == 0)

.forEach(System.out::print);

// MAP EACH NUMBER TO ITS SQUARE

List<Integer> squarelist = numbers.stream()

.map(n -> n \* n)

.collect(Collectors.toList());

System.out.println(squarelist);

// SORT

List<Integer> sorted = numbers.stream()

.sorted()

.collect(Collectors.toList());

System.out.println(sorted);

// OMIT DUPLICATES

List<Integer> noDuplicates = numbers.stream()

.distinct()

.collect(Collectors.toList());

System.out.println(noDuplicates);

// REDUCE

/\* sum of all numbers \*/

int sum = noDuplicates.stream()

.reduce(0, Integer::sum);

System.out.println(sum);

/\* product of all numbers \*/

int product = noDuplicates.stream()

.reduce(1, (n1, n2) -> n1 \* n2);

System.out.println(product);

// MATCH

boolean positive = numbers.stream()

.allMatch(n -> n > 0);

System.out.println(positive);

boolean greaterThan50 = numbers.stream()

.anyMatch(n -> n > 50);

System.out.println(greaterThan50);

boolean negative = numbers.stream()

.noneMatch(n -> n < 0);

System.out.println(negative);

// MAX/MIN

int max = noDuplicates.stream()

.max(Integer::compareTo)

.orElse(0);

System.out.println(max);

int min = noDuplicates.stream()

.min(Integer::compareTo)

.orElse(0);

System.out.println(min);

// COLLECT TO LIST

List<Integer> evenNumbers = noDuplicates.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

System.out.println(evenNumbers);

// SKIP

List<Integer> skipFirstFive = noDuplicates.stream()

.skip(5)

.collect(Collectors.toList());

System.out.println(skipFirstFive);

}

}

Strings

package main;

public class Strings {

public static void main(String[] args) {

// DECLARATION AND INITIALISATION

/\* Direct Initialisation: \*/

String greeting = "Hello,";

/\* Using the "new"-keyword: \*/

String name = new String("John Doe");

// CONCACTENATION

String personalGreeting = greeting + " " + name;

System.out.println(personalGreeting);

// STRING LENGTH

/\* You can get the length of a string using the length() method. \*/

int length = personalGreeting.length();

System.out.println(length);

// ACCESSING CHARACTERS

/\* Syntax: charAt(index) \*/

char firstChar = personalGreeting.charAt(0);

System.out.println(firstChar);

// SUBSTRING EXTRACTION

/\* Syntax: substring(start, end) \*/

String subString = personalGreeting.substring(7, 11);

System.out.println(subString);

// STRING COMPARISON

/\* Syntax: what.equals(with) \*/

String str1 = "Hello";

String str2 = "Hello";

boolean areEqual = str1.equals(str2);

System.out.println(areEqual);

// CONVERSION

String lowerCaseMessage = personalGreeting.toLowerCase();

String upperCaseMessage = personalGreeting.toUpperCase();

System.out.println(lowerCaseMessage);

System.out.println(upperCaseMessage);

// TRIMMING WHITESPACE

String input = " Hello, World! ";

String nextPart = "John is here!";

String trimmedInput = input.trim();

System.out.println(input + " " + nextPart);

System.out.println(trimmedInput + " " + nextPart);

// REPLACING SUBSTRINGS

/\* Syntax: replace(what,with) \*/

String message = "Hello, World!";

String newMessage = message.replace("Hello", "Hi");

System.out.println(newMessage);

// SPLITTING A STRING

/\* Syntax: split(delimiter) \*/

String sentence = "Hello John, John, hello!";

String[] words = sentence.split(" ");

for (String word : words) {

System.out.println(word);

}

// CHECKING PREFIXES

/\* Syntax: startsWith(substring)

endsWith(substring) \*/

boolean startsWith = sentence.startsWith("Hello");

boolean endsWith = sentence.endsWith("hello!");

System.out.println(startsWith);

System.out.println(endsWith);

// CHECKING IF A STRING CONTAINS A SUBSTRING

/\* Syntax: string.contains(substring) \*/

boolean containsHello = sentence.contains("Hi");

System.out.println(containsHello);

// GETTING THE INDEX OF A SUBSTRING

/\* Syntax: string.indexOf(substring) \*/

int index = sentence.indexOf("hello!");

System.out.println(index);

// REVERSING A STRING

StringBuilder reversed = new StringBuilder(sentence).reverse();

System.out.println(reversed);

// JOINING STRINGS

/\* Syntax: join(delimiter, Array) \*/

String[] joinWords = {"Hello", "World", "Java"};

String joined = String.join(" ", joinWords);

System.out.println(joined);

// SPECIAL CHARACTERS

System.out.println("Backslash:" + "\t" + "\t" + "\t" + "\\" + "\\");

System.out.println("Tab:" + "\t" + "\t" + "\t" + "\t" + "\\" + "t");

System.out.println("New Line:" + "\t" + "\t" + "\t" + "\\" + "n");

System.out.println("Quotation Marks:" + "" +

"\t" + "\\" + "\"");

}

}Terminal Operations

package main;

import java.util.Scanner;

public class TerminalOperations {

public static void main(String[] args) {

// READING WORDS

Scanner scanner = new Scanner(System.in);

System.out.print("Enter a string: ");

String text = scanner.next();

System.out.println("You entered: " + text);

// READING LINES

System.out.println("Enter a string: ");

String t = scanner.nextLine();

System.out.println("You entered " + t);

// READING NUMERIC VALUES

System.out.print("Enter an integer: ");

int number = scanner.nextInt();

System.out.println("You entered: " + number);

// READING BOOLEANS

System.out.print("Enter a boolean value (true/false): ");

boolean value = scanner.nextBoolean();

System.out.println("You entered: " + value);

// READING CHARACTERS

System.out.print("Enter a character: ");

char ch = scanner.next().charAt(0);

System.out.println("You entered: " + ch);

// SKIPPING INPUT

System.out.print("Enter a string to skip: ");

scanner.skip("hello");

String txt = scanner.nextLine();

System.out.println("You entered: " + text);

}

}

Testing

package main;

public class Testing {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public double divide(int a, int b) {

if (b == 0) {

throw new ArithmeticException("Cannot divide by zero");

}

return (double) a / b;

}

public boolean isEven(int number) {

return number % 2 == 0;

}

}

package test;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.assertThrows;

public class Testing {

private main.Testing mathUtils;

@BeforeEach

public void setUp() {

mathUtils = new main.Testing();

}

// SIMPLE ASSERTION

@org.junit.jupiter.api.Test

public void testAddition() {

int result = mathUtils.add(2, 3);

assertEquals(5, result);

}

@org.junit.jupiter.api.Test

public void testDivision() {

double result = mathUtils.divide(10, 2);

assertEquals(5, result);

}

// EXCEPTION TESTING

@org.junit.jupiter.api.Test

public void testDivisionByZero() {

assertThrows(ArithmeticException.class, () -> mathUtils.divide(10, 0));

}

// PARAMETERIZED TESTING

@ParameterizedTest

@ValueSource(ints = {1, 2, 3, 4, 5})

public void testIsEven(int number) {

boolean isEven = mathUtils.isEven(number);

assertEquals(number % 2 == 0, isEven);

}

@org.junit.jupiter.api.Test

public void testMultipleOperations() {

mathUtils.add(2, 3);

mathUtils.subtract(10, 5);

mathUtils.multiply(2, 4);

double result = mathUtils.divide(10, 2);

assertEquals(5, result);

}

}