**INTERNITY FOUNDATION**

**TASK-7**

**Submitted By:**

**Amisha Singhal**

**Java Batch**

**Multithreading**

**Ans- Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously. Multiprocessing and multithreading, both are used to achieve multitasking. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.Java Multithreading is mostly used in games, animation, etc.

**Inner Classes**

**Ans-** An inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class.

**Syntax:**

class OuterClass {

...

class InnerClass {

...

}

}

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

**Eg-**

class Outer\_Demo {

int num;

// inner class

private class Inner\_Demo {

public void print() {

System.out.println("This is an inner class");

}

}

// Accessing the inner class from the method within

void display\_Inner() {

Inner\_Demo inner = new Inner\_Demo();

inner.print();

}

}

public class My\_class {

public static void main(String args[]) {

// Instantiating the outer class

Outer\_Demo outer = new Outer\_Demo();

// Accessing the display\_Inner() method.

outer.display\_Inner();}

}

**OUTPUT: This is an inner class.  
Enums**

**Ans-** The **Enum in Java** is a data type which contains a fixed set of constants. The variable must be equal to one of the values that have been predefined for it.

* All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
* **toString() method** is overridden in **java.lang.Enum class**,which returns enum constant name.
* enum can implement many interfaces.
* Enum improves type safety.
* Enum can be easily used in switch.
* Enum can be traversed.
* Enum can have fields, constructors and methods.
* Enum may implement many interfaces but cannot extend any class because it internally extends Enum class.

**Eg-**

**class** EnumExample1{

//defining the enum inside the class

**public** **enum** Season { WINTER, SPRING, SUMMER, FALL }

//main method

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

//traversing the enum

**for** (Season s : Season.values())

System.out.println(s);

}}

**OUTPUT:**

WINTER

SPRING

SUMMER

FALL **Generics and Collections**

**Ans - Generics** mean **parameterized types**. The idea is to allow type (Integer, String etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types.   
An entity such as class, interface, or method that operates on a parameterized type is called a generic entity.

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

The **Collection in Java** is a framework that provides a architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).

**Create and use a generic class**

**Ans-**

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively. Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

A *generic class*in Java is a class that can operate on a specific type specified by the programmer at compile time. To accomplish that, the class definition uses *type parameters*that act as variables that represent types (such as int or String).

To create a generic class, we list the type parameter after the class name in angle brackets. The type parameter specifies a name that we can use throughout the class anywhere otherwise use a type.

**Syntax:**

// To create an instance of generic class

**BaseType <Type> obj = new BaseType <Type>()**

**Eg-**

class Test<T>

{

    // An object of type T is declared

    T obj;

    Test(T obj) {  this.obj = obj;  }  // constructor

    public T getObject()  { return this.obj; }

}

// Driver class to test above

class Main

{

    public static void main (String[] args)

    {

        // instance of Integer type

        Test <Integer> iObj = new Test<Integer>(15);

        System.out.println(iObj.getObject());

        // instance of String type

        Test <String> sObj =

                          new Test<String>("HELLO WORLD!");

        System.out.println(sObj.getObject());

    }

}

**Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects**

**Ans-** ArrayList is an implementation of the List interface that internally uses an Array to store the elements. This implementation is not synchronized.

Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically.

**Eg-**

public class Test {

public static void main(String[] args) {

List<Integer> l = new ArrayList<>();

// Add elements

l.add(1);

l.add(2);

l.add(3);

// Get size.

int count = l.size();

// Get an element with the zero-based index.

Integer i = l.get(0);

}

}

TreeSet is an implementation of the Set interface that uses a tree for storage, which makes access time very fast. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used. This implementation is not synchronized.

Eg- public class Test {

public static void main(String[] args) {

Set<Integer> ts = new TreeSet<>();

// Add elements

ts.add(2);

ts.add(1);

ts.add(3);

// Get size.

System.out.println(ts); // Prints [1,2,3]

}

}

TreeMap is an implementation of the Map interface that uses a tree for storage key/value pairs, which makes access time very fast. The elements are ordered using the natural ordering or their keys, or by a Comparator provided at map creation time, depending on which constructor is used. This implementation is not synchronized.

Eg-

public class Test {

public static void main(String[] args) {

Map<String, Integer> tm = new TreeMap<>();

// Put elements to the map

tm.put("A", 10);

tm.put("C", 40);

tm.put("B", 20);

// Get a set of the entries

Set<Entry<String, Integer>> set = tm.entrySet();

// Get an iterator

Iterator<Entry<String, Integer>> i = set.iterator();

// Display elements

while(i.hasNext()) {

Entry<String, Integer> me = i.next();

System.out.print(me.getKey() + ": ");

System.out.println(me.getValue());

}

// Get an element

Integer i = tm.get("C"));

}

}

ArrayDeque is an implementation of the Deque interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe. Null elements are prohibited. This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces. Elements are stored in the order (first or last) in which they are inserted.

Eg-

public class Test {

public static void main(String[] args) {

Deque<Integer> d = new ArrayDeque();

//Add elements

d.add(1); //add element at tail

d.addFirst(2); //add element at head

d.addLast (3); //add element at tail

//Get elements

Integer firstElement1 = d.element(); //peek at the element at the head without taking the element out of the queue (throws exception is the queue is empty)

Integer firstElement2 = d.peek(); //peek at the element at the head without taking the element out of the queue (returns null is the queue is empty)

Integer firstElement3 = d.getFirst();//get first element (throws exception is the queue is empty)

Integer firstElement4 = d.peekFirst();//get first element (returns null is the queue is empty)

Integer lastElement1 = d.getLast();//get last element (throws exception is the queue is empty)

Integer lastElement2 = d.peekLast();//get last element (returns null is the queue is empty)

//Remove elements

Integer element1 = d.remove(); //retrieves and removes the head of the queue

Integer element2 = d.removeFirst(); //retrieves and removes the first element of the queue

Integer element3 = d.removeLast(); //retrieves and removes the last element of the queue

}

}

**Use java.util.Comparator and java.lang.Comparable interfaces**

**Ans-**

import java.util.Comparator;

public class Employee implements Comparable<Employee> {

private int id;

private String name;

private int age;

private long salary;

public int getId() {

return id;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public long getSalary() {

return salary;

}

public Employee(int id, String name, int age, int salary) {

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

@Override

public int compareTo(Employee emp) {

//let's sort the employee based on an id in ascending order

//returns a negative integer, zero, or a positive integer as this employee id

//is less than, equal to, or greater than the specified object.

return (this.id - emp.id);

}

@Override

//this is required to print the user-friendly information about the Employee

public String toString() {

return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age + ", salary=" +

this.salary + "]";

}}

**OUTPUT:**

Default Sorting of Employees list: [[id=1, name=Pankaj, age=32, salary=50000], [id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000]]

**Collections Streams and Filters**

**Ans –** Collections are primarily concerned with the efficient management of, and access to, their elements.Streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source. However, if the provided stream operations do not offer the desired functionality, the [BaseStream.iterator()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#iterator--) and [BaseStream.spliterator()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#spliterator--) operations can be used to perform a controlled traversal.

The filter() method takes a Predicate as parameter. The Predicate interface contains a function called test() which the [**lambda expression**](http://tutorials.jenkov.com/java/lambda-expressions.html) passed as parameter above is matched against. In other words, the lambda expression implements the Predicate.test() method.

**Syntax :**

**Stream<T> filter(Predicate<? super T> predicate)**

**Eg stream.filter( item -> item.startsWith("o") );**

**Iterate using forEach methods of Streams and List**

**Ans-** ForEach states that it “performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.”

With forEach, we can iterate over a collection and perform a given action on each element, like any other Iterator.

**Eg-**

List<String> names = Arrays.asList("Larry", "Steve", "James"); names.forEach(System.out::println);

**Eg-**

List<String> alphabets = new ArrayList<>(Arrays.asList("aa", "bbb", "cat", "dog")); alphabets.forEach(s -> System.out.println(s));  
  
 **Describe Stream interface and Stream pipeline**

**Ans-** Stream is a sequence of elements from a source supporting sequential and parallel aggregate operations.

Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

Intermediate operations return a new stream. They are always lazy; executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.

Terminal operations, such as Stream.forEach or IntStream.sum, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used; if you need to traverse the same data source again, you must return to the data source to get a new stream. In almost all cases, terminal operations are eager, completing their traversal of the data source and processing of the pipeline before returning. Only the terminal operations iterator() and spliterator() are not; these are provided as an "escape hatch" to enable arbitrary client-controlled pipeline traversals in the event that the existing operations are not sufficient to the task.

**Stream Pipelines**

To perform a computation, stream operations are composed into a **stream pipeline**. A stream pipeline consists of:

* a source (which might be an array, a Collection, a generator function, an I/O channel, etc.)
* zero or more intermediate operations (which transform a Stream into another Stream, such as filter(Predicate p))
* a terminal operation (which produces a result or side-effect, such as count() or forEach(Consumer c))

Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

**Filter a collection by using lambda expressions**

**Ans-** The Stream.filter() methods is an intermediate operation. It filters a stream on the basis of a given predicate and returns a stream object. And again you can apply other stream methods to this instance. The method syntax is as follows

Stream<T> filter(Predicate<? super T> predicate)

Eg- An Employee list and created a Predicate. This predicate will be passed to filter as an argument. Finally we will print the filtered stream:

public class Employee {

public String name;

public double salary;

public Employee(String n, double s) {

name = n; salary = s;

}

public String toString() { return name + " : " + salary; }

}

List<Employee> emps = new ArrayList<>();

emps.add(new Employee("John", 120000.0));

emps.add(new Employee("Daniel", 112000.0));

emps.add(new Employee("Dzmitry", 36000.0));

emps.add(new Employee("Steven", 150000.0));

Predicate<Employee> p = emp -> emp.salary > 100000.0;

Consumer<Employee> c = emp -> System.out.println(emp);

emps.stream().filter(p).forEach(c);

**OUTPUT:**

John : 120000.0

Daniel : 112000.0

Steven : 150000.0

**Use method references with Streams**

**Ans-** Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

**Eg-**

myList

.stream()

.filter(s -> s.startsWith("c"))

.map(String::toUpperCase)

.sorted()

.forEach(System.out::println);

**========================END===============================**