**JAVA**

* extension .java
* javac filename.java-------->compilation
* java filename-------> for execution
* if you do any changes in the program then you want to reflect that changes in the execution then we go for re-compilation
* JRE takes the bytecode converts machine language after converting the code will be executed.
* JRE----> interpreter.
* 0’s and 1’s ------🡪 low level language.
* Java is a strongly typed language

“Java is an **object oriented programming** language.”

**OBJECT:**

Objects are elements of the program that has some data which is also known as state.

Objects also have behaviours which means they can perform certain operations. Those behaviours in java is known as methods.

States--------> data

Behaviour---->methods

**KEY WORDS**

Keyword is a name has some predefined meaning in the programming language.

In java there are 53 keywords, in those three are literals(true, false these literals also have some predefined meaning so those are also considered as keywords).

**IDENTIFIERS**

Identifier is a name given to components/elements of a program

Rules to declare identifier:--

* Always starts with alphabets
* It can not start with a digit. But it can have n no.of digits after declaring first character as alphabet.
* It can not support white spaces.
* Identifiers also case sensitive.
* We can not declare keywords as identifiers.
* It can not support special characters except $ and \_.

**Note**:-- A java file can contain multiple classes but the public class name should be same as file name.

**VARIABLES**

JDK ---> we want to develop an application

JRE---> we want to executes an application

Variable is a name given to a memory. Variable can contain memory to store data. Every variable should have data type.

Variables

Local Global

Primitive reference primitive reference

Static non-static static non-static

**DATA TYPES**

Data type is used to specify the type of the data it should store in memory.

Types of data types:--

1.Primitive

2.Nonprimitive/reference type.

1. Primitive :--

We have 8 primitive types in java

1. byte(1byte) --------------> -128 to 127
2. short(2bytes) -------------> -32768 to 32767
3. int(4 bytes) --------------> -2 million to 2 million
4. long (8bytes) -------------> -9 billion to +9 billion
5. float (4bytes)
6. double (8bytes)
7. char
8. boolean (true/false).

Datatype var ; // declaration

Var = value; //initialization

System.out.println(var); //utilization

**OPERATORS**

An operator is a special symbol or keyword that is used to designate a mathematical operation or some other type operation.

These operations can we perform on one or more than one values called as an operands.

Arithmetic Operators :--

+ adds two operands

- subtract right operand from the left operand

\* multiply two operands

/ divide the left operand by right operand

% it will return the remainder value.

Assignment Operator:---

= assign a value to a variable.

Combination of arithmetic and assignment is known as compound assignment operators.

Ex:-- +=,-=, \*=, /=, %=

Increment/Decrement Operators :---

We can call this operators as unary operators.

Increment(++)

Decrement(--)

Increment

Pre-increment(++i) Post-increment(i++)

(i). increment (i).substitute

(ii). Substitute (ii).utilization

(iii).Utilize (iii).increment

In pre-increment first it will increment the value by 1 and assign that incremented value to the variable.

In post-increment first it will assign the value then increment the value by 1

Relational Operators/comparison Operators :----

These operators are used to relation between two operands

== It return true if both sides are equal otherwise return false.

!= return true both sides of the equation are not equal otherwise return false

< returns true if the left side of the equation is less than right side of the equation.

> returns true if the left side of the equation is greater than right side of the equation.

>= returns true if the left side of the equation is greater than or equal to right side of the equation.

<= returns true if the left side of the equation is less than or equal to right side of the equation.

Logical Operators:--

We have three types of logical operators

(i). NOT (!) : It is a unary operator it returns false if the RHS operator is true

(ii). AND (&&) : It returns true if both the conditions are true otherwise return false

(iii). OR (||) : It will returns true if any of the condition is true otherwise returns false.

Bitwise Operators :--

Performs the operation bit by bit

&(if both are 1 then return 1)

0010

0011

-----------------------

0010

| ( any of one is 1 return 1)

0010

0011

-----------------------

0011

^ ( if both inputs are match then return 0)

0010

0011

-----------------------

0001

**CONTROL FLOW STATEMENTS**

By using this we can control the flow of execution of the program.

1.if :--

condition

TRUE

Code/statements inside of block

False

Code/stmts after if block

If-else:--

Boolean condition

false true

Code out side of the if else block

Code inside if block

Code inside else block

If-elseif:--

We go for elseif when we have more than one boolean condition to be satisfied. Whenever we have multiple conditions we go for elseif.

Syntax:

If(condition)

{

--------------

------------------

}

elseif(Condition){

----------------

------------------

}else{

------------------

------------------

}

Switch:--

Condition

Case1

Case2

Case3

Code outside switch

**Looping statements:**

For:--

Syntax:--

for (initialization; condition; Increment/decrement){

-------------------

--------------------

}

Code outside for loop

Inc/dec

Code inside for loop

initialization

condition

false true

class ForLoop{

public static void main(String[]args){

int i=1;

for( ; ; ){

System.out.println("i = "+i);

i++;

}

System.out.println("outside for");

}

}

OUTPUT:----

C:\Users\sumav\Desktop\java>javac ForLoop.java

ForLoop.java:8: error: unreachable statement

System.out.println("outside for");

^

1 error.

**While :---**

Syntax :--

While(condition){

Statements…….

Statements…….

}

**do-while :---**

do{

statements………

statements………

}while(condition);

**Methods**

Methods can perform specific task

Syntax :---

Access specifier Access modifiers Return Type method name(arg list){//declaration

Statements…………. //definition

Statements………….

}

Access specifies :-- Controls the visibility of the any of the code

Access Modifier :--

Return type :-- Return can any of the primitive data type or void or any of the

reference type.

A method will only execute if it has been called by passing the required arguments.

A method which is being called is known as called method.

A method which is calling another method is known as calling method.

We can not create a method inside another method.

**ARRAYS**

Array is a group of homogeneous data that has some index and a fixed size .

Index of an array always starts with 0 and ends with size-1.

 the size of an array is fixed and cannot increase to accommodate more elements.

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Syntax to declare array:----

Datatype []arrayname;

Datatype[] arrayname;

Datatype arrayname[];

Creation of an array:--

intArr = new int[size];

(or)

Int[] intArr=new int[size];

Initialization :---

intArr[index]=value;

we can find the length or size of an array by using a variable “length”.

**STRING:---**

String is a sequence of characters. Internally it is using character array to store the data.

String is immutable

Immutable:--

If we want to make any changes in the string we can not make the changes in the same string we have to create a new string.

Syntax: --

String str = ”hello”;

String str;

String str=new String(“Hello”);

String methods:--

String a=str.substring(10);

System.*out*.println(a);

In this the method will take only one argument that is the index of the element.

a=str1.substring(4, 11);

System.*out*.println(a);

ending index elements are excluded.

Reference type :--

A reference type is a type that is based on class rather than a primitive data type.

The reference type can be based on predefined classes in java or classes defined by programmer or developer.

Static and non static:---

Static:--

In any member of the class that have been declared by using the keyword static is static member.

If we want to use static methods in some other class we need not to create an object for the class which is having static members, directly we can use by using class name and dot(.) operator.

We can declare static members inside the class and outside of the class.

Non-static:--

Members which does not contain static keyword is known as non static members(methods or variables).

We can access those non static members in outside of the class we need to create an object for that particular class.

**Constructor**

Constructor is a special type of method.

Constructor name should be same as class name.

Constructor does not have any return type.

Every class must and should have a constructor.

If a class does not contain any constructor then the java compiler will add the default constructor for that class. Default constructor is created by compiler.

Whenever we are creating an object of a class the constructor will be invoked.

If constructor is taking any input data then we can call it as parameterized constructor.

The constructor does not contain any arguments is known as no-argument constructor. It is created by programmer.

Constructor overloading: ----

We are trying to create a same constructor but different arguments in the class is known as constructor overloading.

A class contains more than on constructor with same name but change in the type of the arguments or by change in the no.of arguments

**EXAMPLE:-**

**public** **class** ConstructorExample {

**public** ConstructorExample(**int** i){

System.*out*.println("This is const with int arg");

}

**public** ConstructorExample(){

System.*out*.println("this is no-arg constructor");

}

**public** ConstructorExample(String s){

System.*out*.println("This is const with String arg");

}

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

ConstructorExample c=**new** ConstructorExample();

ConstructorExample c1=**new** ConstructorExample(5);

ConstructorExample c2=**new** ConstructorExample("suma");

}

}

If we want to use methods and variable present in another class which is present in another package we can make the use of import statement.

**Access Specifiers**: ---

**Public:** it can be used by any other class.

**Protected:** it can be accessed within the same package we can not

access in another packages and subclasses of another package.

**Default/package level**: we can not use methods in some other

packages

**Private:** the scope of this access specifier is within the class.

**Association :--**

Association in java is relationship between two classes.

Types of association:

One-to-one

One-to-many

Many-to-many

Many-to-one

In java we can have two types of relationships.

1.is-a relation

2.has-a relation

1.has- a relation: -

Aggregation:

one class is not depending upon it is called as aggregation.

Composition:

It is a kind of has-a relationship. Existence of one class will depend on another class.

2.is-a relation:

It is known as inheritance.

**INHERITANCE**

The process by which one class acquires the properties and functionalities of another class is called as inheritance.

The aim of inheritance is to provide reusability of code.

The class whose properties or functionalities are being inherited to other class is known as parent class or super class or base class.

The class which is acquiring the properties or functionalities from other class is known as child class or sub class or derived class.

For inheritance in java we have to make use of “extends” keyword

You can access the properties of both super class and subclass using object of subclass.

Final classes cannot be inherited.

Final members of a super class can be inherited but cannot be change.

Private members and constructors of super class cannot be inherited.

Types of inheritance: ---

1.single inheritance:

One class is acquiring the properties from another class is known as single inheritance.

parent

child

Multi level inheritance:

In multi-level inheritance a class is extending from another class which is already extended from some another class.

A

B

C

Multiple inheritance: ---

Java does not support multiple inheritance.

Hierarchical inheritance: --

Two or more classes extending from one class is known as hierarchical inheritance.

Hybrid inheritance: --

It is a combination of two or more inheritance.

**Super: --**

Super keyword refers to the immediate superclass data members.

To access the data member of parent class when both child and parent class have the data member with the same name.

Syntax: --

Super.datamember;

If sub class constructor does not contain super() then java compiler implicitly adds super() in subclass constructor.

Java compiler implicitly adds super() when the super class contains no-arg constructor otherwise we have to pass the super() explicitly in subclass.

**Method Overriding**

Declaring a method in child class which has already being declared in parent class is known as method overriding.

Method overriding is done to provide implementation specific to a child class.

Overridden method is present in parent class

Overriding method is present in child class

Advantages: ---

Advantage of method overriding is that we can provide implementation to the child class method without changing the code present in the parent class.

Whenever we need to override a method we need to give a @Override .

We can not override static, private methods that are present in super class.

Final members can be inherited and we can use those final members but we cannot modify final members.

**Method Overloading: --**

Method overloading is feature in java that allows us to have same methods with the same name in a single class more than once.

Provide the arguments differ

1.no.of arguments.

2.order of arguments.

3.data type of an arguments.

WAP that performs basic arithmetic operations having different argument list.

**POLYMORPHISM**

Polymorphism is an ability in which one entity shows behaviours of another entity.

Method overriding is a type of polymorphism.

Method overriding is an example of “runtime polymorphism.”

Method overloading is an example of “compile time polymorphism.”

Runtime polymorphism/late binding

Compile time polymorphism/early binding.

**Abstraction**

Hiding the implementation details and providing the necessary functionalities to the user is known as a “abstraction”.

We can achieve abstraction in two ways

1. by using Abstract class

2. by using interface

Abstract class: --

Any class which is declared with the keyword abstract is known as abstract class.

Any method which is declared with the keyword abstract is known as abstract method.

Abstract methods do not have body, abstract methods end with semicolon.

Abstract classes can have both abstract methods as well as concrete methods.

Any class having an abstract method should be declared as Abstract class.

Any class we can declared as an Abstract class.

If a class has been declared as abstract then that class should be extended in subclasses.

We cannot create an object for abstract class but it can contain constructor.

If any class is extending from abstract class then all the methods present in abstract class should be implemented in subclass otherwise subclass becomes an abstract class.

**Interface**

By default all the methods present in interface are abstract.

Interface does not contain a constructor.

We can implement more than one interface at a time.

If we declare an annotation as @FunctionalInterface then we can declare only one method in interface.

An interface which does not contain any methods it is called marker interface

Ex: --Cloneable, Serializable, Remote.

\*\* All the methods, classes, interfaces present in java. lang package by default available to all java classes.

Differences between abstract class and interface

**Abstract class** **Interface**

1. An Abstract class is defined by the 1.An Interface is defined by

Keyword abstract. The keyword Interface.

2.Abstract class contains both abstract 2. By default all the methods methods and concrete method present in interface are abstract.

3.Abstract class can contain constructor 3.Interface does not contain even though we cannot create an object constructor.

4. By default all the variable present in 4. By default all the variables Abstract class not static, final present in interface static, final.

5.We need not to perform initialization for 5. We should perform

All the variables at the time of declaration initialization for the variable at the time of declaration.

6. We can use extends keyword to implement 6. We can use “implements”

abstract methods in subclasses to implement abstract methods in subclass

**Encapsulation**

Binding the data and code into a single unit is known as encapsulation.

Encapsulation is a mechanism with which we wrapper the data members and function members into a single object.

All data members present in the class are private we can access those private data members with the help of two methods

1.setter

2.getter

**Package**

Whenever we are using two classes with same but both are present in different packages then we have to give fully qualified class name for any of the one class.

**package** com.dev. methods;

**import** com.dev.encapsulation.Dog;

**public** **class** Demo {

Dog d=**new** Dog(); //It is present in encapsulation package

com.dev.constructors.Dog d1=**new** com.dev.constructors.Dog();

}

**Final: -----**

If we declare the variable as final we cannot change/reinitialize the value of the variable.

If we declare a method as final we cannot override it in the subclasses.

If we declare a class as final we cannot extend it into subclasses.

Final classes can not be the super classes but can be the subclasses.

**Object**

In java each and every class directly or indirectly inherits the properties of Object class i.e., Object class is the super most class in java.

Each and every class either a predefined or user defined class is a child class of object.

**Methods of Object class: ---**

1**.** getClass(): --

2. clone(): --

3. equals():

4. toString():

5. wait():

6. wait(long):

7. wait(long, int)

8. notify():

9. notifyAll():

10. finalize():

11. hashCode()

**Strings**

Strings are reference types. As a result a string variable holds a reference to an object related from the String class, not the value of the string itself.

Even though strings are not primitive types, the java compiler has some features designed to let you work with strings almost as they were primitive types.

You can combine, or concatenate, strings by using ‘+’ operator.

String is immutable. An immutable object is an object, once we create it we cannot change it.

The string pool contains two parts. (i) Constant pool. (ii) Non-constant pool. Within the constant pool duplicates are not allowed. Whereas within non-constant pool duplicates are allowed.

**StringBuilder and StringBuffer classes :**

The String class is powerful, but it is not very efficient for programs that require heavy-duty string manipulation. Because String objects are immutable, any method of the String class that modifies the String in any way must create a new String object.

To overcome this problem java offers two alternatives to the String class: StringBuilder and StringBuffer.

To StringBuilder and StringBuffer classes are mirror images. Both have the same methods and perform the same String manipulation.

The only difference between StringBuffer and StringBuilder is StringBuffer is thread safe whereas StringBuilder is not thread safe.

**Constructors: --**

**i. StringBuffer: ---**

1.

Constructs a string buffer with no characters in it and an

\* initial capacity of 16 characters.

Public StringBuffer(){

Super(16);

}

2.

Constructs a string with no characters in it and the specified initial capacity.

Public StringBuffer(int initial Capacity){

Super(capacity);

}

3. Constructs a string buffer initialized to the contents of the

\* specified string. The initial capacity of the string buffer is

\* <code>16</code> plus the length of the string argument.

public StringBuffer(String str) {

super(str.length() + 16);

append(str);

}

4.

public StringBuffer(CharSequence seq) {

this(seq.length() + 16);

append(seq);

}

**ii. StringBuilder: --**

**1.**

public StringBuilder() {

super(16);

}

2.

public StringBuilder(int capacity) {

super(capacity);

}

3.

public StringBuilder(String str) {

super(str.length() + 16);

append(str);

}

4.

public StringBuilder(CharSequence seq) {

this(seq.length() + 16);

append(seq);

}

|  |  |  |
| --- | --- | --- |
| **String** | **StringBuffer** | **StringBuilder** |
| 1.String is immutable | 1. Mutable | 1. Mutable |
| 2. It has 16 Constructors | 2.It has 4 Constructors | 2.it has 4 constructors. |
| 3. we can assign string directly | 3. We can assign string with the help of new Keyword | 3. We can assign string  with the help of new  keyword |
| 4.fixed length | 4. Growable length | 4. Growable length |

**Exception: --**

Exception is an error event that happens during the execution of the program and disturbs the normal flow of execution.

**Exception Handling: --**

Exception in java can arise from different kinds of situations such as wrong data entered by user, hardware failure, network connection failure, Database server down etc.

Whenever an error occurs while executing a statement, creates an exception object and then the normal flow of the program halts and JRE tries to find someone that can handle the raised exception.

The exception object contains a lot of information such as method hierarchy, line number where the exception occurred, type of exception etc.

The process of creating the exception object and handing it over to runtime environment is called “throwing the exception”.

If appropriate exception handler handler is found, exception object is passed to the handler to process it. The handler is said to be “catching the exception”.

Q. Differentiate exceptions and errors.

|  |  |
| --- | --- |
| Error | Exception |
| Errors occurs when we make a syntactical mistake. | Exceptions occurs when we make a mistake in logic. |
| Errors occurs at compile time | Exceptions occurs at runtime |
| Errors are irrecoverable | Exceptions are recoverable |

Java Exception hierarchy: --

Java Exceptions are hierarchical and inheritance is used to categorize different types of exceptions.

Throwable is the parent class of java Exceptions hierarchy and it has two child objects---Error and Exception

**Errors: ---**

Errors are exceptional scenarios that are not of application and its not possible to anticipate and recover from them.

For example Stack Overflow Error, Hardware Error.

**Exception: --**

**Collection Framework**

Collections are like containers that group multiple items in a single unit.

Collections are used almost in every programming language and when java arrived, it also came with few Collection classes: Vector, Stack, HashTable, Array.

Java 1.2 provided Collections Framework that is architecture to represent and manipulate Collections in java in a standard way.

Interfaces:

Java Collection Framework interfaces provide the abstract data type to represent Collection. Java.util.Collection is the root interface of Collection Framework.

It contains some important methods such as size(), iterator(), add(), remove(), clear().

Some interfaces are List, Set, Queue, Map.

Implementation classes:

Collections in java provides core implementation classes for collections. We can use them to create different types of collections in java program. Some important collection classes are : ArrayList, LinkedList, HashMap, TreeMap, HashSet.

All the core collection interfaces are generic.

Ex: -- public interface Collection<E>.

When we declare collection, we should use it to specify the type of object it can contain.

It helps to reducing run-time errors by type checking the objects at compile time.

**Collection interface: --**

This is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The java platform does not provide any direct implementation of this interface.

The interface has methods to tell you how many elements are in the collection(size, isEmpty), to check whether a given object is in the collection(contains), to add and remove an element from the collection(add, remove).

containAll, removeAll

**Set**

Set is a collection that contains no duplicate elements. Set contain no pair of elements and at most one null element. This interface models the mathematical set abstraction and is used to represent sets such as deck of cards.

Java platform contains three general-purpose set implementations: HashSet, TreeSet, LinkedHashSet.

**List**

List typically allows duplicate elements.

The user of this interface has precise control over where in the list each element is inserted.

More formally, lists typically allow pairs of elements <tt>e1</tt>

List interface provides useful methods to add an element based on the index, remove/replace element.

**Queue interface(FIFO):**

A collection designed for holding elements prior to processing. Besides basic collection operations, queues provide additional insertion, extraction, inspection.

Boolean offer(E e): --

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.

It returns true if the element was added to this queue, else return false.

E poll(): --

Retrieves and removes the head of this queue, or returns null if this queue is empty.

E element():

Retrieves, but does not remove, the head of this queue. This method differs from peek() only in that it throws an exception if this queue is empty.

If this queue is empty then it will throws an exception as NoSuchElementException.

**Map: --**

Java Map is an object that maps keys to values. A map can not contain duplicate keys: Each key can map to at most one value.

The java platform contains three general-purpose Map

implementations: HashMap, TreeMap, LinkedHashMap

The basic operations of Map are put, get, containsKey, containsValue, size and isEmpty.

**HashSet: --**

It is the basic implementation of the Set interface that is supported by a HashMap. It makes no guarantee for iteration order of the set and permits the null element.

The basic operations are: add, remove, contains, size.

We can set the initial capacity and load factor is a measure of how full the HashMap is allowed to get before its capacity is automatically increased.

Constructors: ---

1. public HashSet(){

Map=new HashMap<>();

}

Constructs a new empty set

2. public HashSet(Collection <? Extends E> c){

Map=new HashMap<>(Math.max(int) (c.size()/0.75)+1,16));

}

3. public HashSet(int initialCapacity, float loadFactor){

Map=new HashMap<>(initialCapacity, loadFactor);

}

loadFactor: -- the loadFactor of the hash map.

4. public HashSet(int initialCapacity){

Map=new HashMap<>(initialCapacity);

}

5. public HashSet(int initialCapacity, float loadFactor, Boolean dummy){

Map=new LinkedHashMap<>(initialCapacity, loadFactor);

}

Constructs a new empty linked hash set. The backing HashMap instance is a LinkedHashMap with the specified initial capacity and the specified load factor.

**HashMap: --**

Basic operations are get and put.

**TreeSet: --**

Constructors

1. TreeSet(NavigableMap<E,Object> m) {

**this**.m = m;

}

Constructs a set backed by the specified navigable Map.

2. public TreeSet() {

this(new TreeMap<E,Object>());

}

Constructs a new, empty tree set, sorted according to the natural ordering of its elements. All elements inserted into the set must implement the Comparable interface.

3.public TreeSet(Comparator<? super E> comparator) {

this(new TreeMap<>(comparator));

}

4. public TreeSet(Collection<? extends E> c) {

this();

addAll(c);

}

5. public TreeSet(SortedSet<E> s) {

this(s.comparator());

addAll(s);

}

**ArrayList: --**

Constructors

1. public ArrayList(int initialCapacity) {

super();

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal Capacity: "+

initialCapacity);

this.elementData = new Object[initialCapacity];

}

Constructs an empty list with the specified initial capacity. initialCapacity the initial capacity of the list IllegalArgumentException if the specified initial capacity is negative.

2. **public** ArrayList(Collection<? **extends** E> c) {

elementData = c.toArray();

size = elementData.length;

// c.toArray might (incorrectly) not return Object[] (see 6260652)

**if** (elementData.getClass() != Object[].**class**)

elementData = Arrays.*copyOf*(elementData, size, Object[].**class** }

3. **public** ArrayList() {

**this**(10);

}

Methods:

**1. public** **void** trimToSize() {

modCount++;

**int** oldCapacity = elementData.length;

**if** (size < oldCapacity) {

elementData = Arrays.*copyOf*(elementData, size);

}

}

Trims the capacity of this ArrayList instance to be the list's current size. An application can use this operation to minimize the storage of an ArrayList instance.

**2. public** **void** ensureCapacity(**int** minCapacity) {

**if** (minCapacity > 0)

ensureCapacityInternal(minCapacity);

}

Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. minCapacity the desired minimum capacity

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Collection | Ordering | Key Value | Duplicate Elements | Null Elements | Thread Safe | LoadFactor | Initial capacity |
| ArrayList | Yes | No | Yes | Yes | No |  | 10 |
| LinkedList | Yes | No | Yes | Yes | No | 0.5 |  |
| HashSet | No | No | No | Yes | No | 0.75 | 16 |
| TreeSet | Yes | No | No | No | No |  |  |
| HashMap | No | Yes | No | Yes | No | 0.75 | 16 |
| TreeMap | Yes | Yes | No | No | No |  |  |
| HashTable | No | Yes | No | No | Yes | 0.75 | 11 |
| Vector | yes | No | yes | yes | yes |  | 10 |

**Java Lambda Expressions: --**

Lambda expression is a new feature of java which was introduced in java SE 8.

The lambda expression is used to provide the implementation of a functional interface. It saves a lot of code.

In case of lambda expression, we don’t need to define the method again for providing the implementation.

Why use Lambda Expression: --

1. To provide the implementation of functional interface

2. Less coding.

Syntax: ---

(argument-list) -->{body};

Java lambda Expression is consisted of three components.

**Threads**

A thread is a single sequence of executable code within a larger program. All the programs so far have used just one thread that is main thread that starts automatically when you run the program—but java lets you create programs that start additional threads to perform specific task.

Ex: -- 1. Web browsers can download files while letting you view web pages. When you download a file in a web browser the browser starts a separate thread to handle the download.

2. Word processor can also check your spelling as you type. Depending on how the word processor is written, it may run the spell checks in a separate thread.

Multithreading: --

Thread creation: --

In java threads can create in two ways

1.by extending Thread class

We create a class that extends java.lang.Thread class. This class overrides the run() available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

2. by implementing the Runnable interface.

We can create a class which implements Runnable interface and override run() method. Then we instantiate a thread object and call start() method on this object.

Every thread will have three important properties: --

1. Thread Name

2. Thread Id

3. Thread Priority

Thread Name: -- Thread Name can be created by programmer in order to identify the threads.

Thread life Cycle: --

The start method creates the system resources, necessary to run the thread, schedules the thread to run, and calls the thread’s run method.

notify(), notifyAll() wait()

R Running

start() join() yield() stop()

sleep()

A thread dies naturally when the run() exits.

A thread can be in one of the following states.

1. NEW

2.RUNNABLE

3.BLOCKED

4. WAITING

5. TERMINATED

Race condition: -- Race condition occurs in a multi-threaded environment when more than one thread try to access a shared resource at the same time.

Synchronization: --

In many cases concurrently running threads share data and two threads try to do operations on the same variable at the same time. This often results in corrupt data as two threads try to operate on the same data.

A popular solution is to provide some kind of lock primitive. Only one thread can acquire a particular time. This can be achieved by using a keyword “synchronized”.

By using synchronize only one thread can access the method at a time and a second call will be blocked until the first call returns or wait() is called inside the synchronized method.

Difference between run() method and start() method

|  |  |
| --- | --- |
| Start() | Run() |
| start() methods only schedules the thread for execution and not actually begins the execution of the thread | The execution of the thread is started when the JVM calls the run() method |
| The **start()** method of thread class is used to begin the execution of thread | if you call run() method directly **no new Thread is created** and code inside run() will execute on **current Thread**. |

Difference between sleep() and wait() methods

|  |  |
| --- | --- |
| Wait() | Sleep() |
| wait() releases the lock or monitor | sleep() doesn’t releases the lock or monitor while waiting. |
|  |  |
|  |  |

**RegEx:**

**https://github.com/rohitjaishwal/JAVA-RegEx**