```
public class MultiThreadDemo {
    public static void main(String[] args)
    {
     MultiThreadTwo  m = new MultiThreadTwo();
     Thread t = new Thread(m);
     t.start();
     Syso("main thread is started");
     for(int i=0; i<10; i++)
      {
      Syso("main thread");
      }
    }// end of main()
}
```

join():-

If a Thread wants to wait until completing some other Thread then we should go for join().

example:-

```
package com.rameshsoft.multithreading;
public class MultiThreadDemo
{ public static void main(String[] args) throws InterruptedException
      {
       MultiThreadOne  m = new MultiThreadOne();
       Thread t =new Thread(m);
```

```
        t.start();
        t.join();
        Syso("main thread is started");
        for(int i=0; i<10; i++)
        {
        System.out.println("main thread");
        }
    }
}
class MultiThread implements Runnable{
  public void run()
  {
   Syso("child thread is started");
    for(int j=0; j<10; j++)
      {
       Syso("child thread");
      }
   }//run()
}//end of class
```

sleep() :- Syntax :- public static final void sleep(long msec)

If a thread don't want to perform any operation for particular amount of time then go for sleep().

Example :-
```
public class SleepDemo{
    public static void main(String[] args){
        Syso("Main Thread");
        Thread.sleep(2000);
        Syso("End of the program");}}}
```

Inner Classes

A class inside another class is called inner class.

→ Like methods, variables of a class even we can have a class as well.

→ The class within is called nested class, and the class that holds the inner class is called outer class.

Syntax:-

```
class Java-OuterClass {
    =
    class Java-InnerClass
        {
        =
        }
}
```

Q: Why we use nested classes?

① It is a way of logically grouping classes that are only used in one place.

② Nested classes represent special type of relationship i.e, it can access all data members and methods of Outer class including private.

③ we can get code optimization.

Types of nested classes:

Nested classes are two types.

① inner classes
   ⓐ Local inner class
   ⓑ Anonymous inner class
   ⓒ member inner class

② static nested classes.

Inner class :-

creating a class inside another class is called inner class.

It is also called as Java member inner class.

→① We can declare inner class as a private but not outer class as a private.

→② when we declare inner class as private, it cannot be accessed from an object of outside the class and we can access through a method.

Example:-

```
class OuterDemoOne {
        String name = "RameshSoft";
            //inner class
            private class InnerDemo {
                    public void display()
                    { Syso("This is a inner class"); } }
```

```
//Accessing the inner class from the method within
    void display_Inner(){
        InnerDemo inner = new InnerDemo();
        inner.display_Inner();
    }
}
public class InnerPrivateTest {
    public static void main(String[] args)
    {   //instantiating the outer class
        OuterDemoOne outer = new OuterDemoOne();
        //Accessing the display_Inner()
        outer.display_Inner();
//OuterDemoOne.InnerDemo inner = outer.new InnerDemo(); //CE
                                    ; change visibility of inner class
    }
}
```

→③ Accessing private variables and methods of outer class inside inner class.

ex:

```
class OuterDemo{
    //private variable of the outer class
    private int first;
    public void initialize(){
    @SuppressWarnings("resource")
```

```
Scanner scanner = new Scanner(System.in);
Syso("enter a number");
first = scanner.nextInt();
}
public class InnerDemo {
  public String getName(String name){
     Syso("inner class method");
      return name;
    }
  public int getOuterNum()
    { initialize();
     return first;
    }
  }
}
public class MyClassDemo{
  public static void main(String[] k){
    // instantiating the outer class
     OuterDemo.InnerDemo inner = outer.new InnerDemo();
     Syso(inner.getName("rameshsoft"));
     Syso(inner.getOuterNum());
      }
}
```

Method Local inner class :—

we can create a class inside the method is called method local inner class.

The scope of inner class is restricted to within the inner class.

Example :—

```
public class MethodClass {
    //instance method of the outer class
    void hello(){
        int first = 56;
        //method-local inner class
        class MethodInner_Demo{
            public void print()
            { Syso("This is method inner class "+first);}
        }//end of inner class
        //Accessing the inner class
    MethodInner_Demo inner = new MethodInner_Demo();
        inner.print();
    }
public static void main(String[] k)
  {
   MethodClass outer = new MethodClass();
       outer.hello();
   }
}
```

Anonymous inner class:—

An inner class declared without a class name is known as Anonymous inner class.

* In case of Anonymous class, we declare and instantiate them at the same time.

Syntax:

```
AnonymousInner anonymous = new AnonymousInner()
    {
      public void m1(){
                =
      };
    }
```

* Generally, we use anonymous class whenever we need to override the method of class or interface.

Example 1:—

```
interface Test
{
  void hello();
}
public class AnonymousTest {
    public static void main(String[] k) {
        Test test = new Test() {
              public void hello() { Syso("Example of anonymous");
            } };
        test.hello(); } }
```

Example 21—

```
abstract class Test
 {
  public abstract void hello();
 }
public class AnonymousTest{
   public static void main(String[] k)
       {
          Test inner = new Test(){
                    public void hello(){ sysout("Example of anonymous");
                                        }
                  };
         inner.hello();
   }
}
```

~~Static Nested class~~

P.T.O

Static Nested class :-

> static is a keyword, which we can apply for variables, methods and for inner classes.

> A static inner class is a nested class which is a static member of outer class.

> It can be accessed without instantiating the outer class.

How to define static nested class?

syntax:

```
class OuterClass
  {
     static class InnerClass
          {
          =
          }
     =
  }
```

how to access static nested class?

syntax: OuterClass.InnerClass varname = new OuterClass.InnerClass();

or

Or

```
public abstract class By {

    public static By id(final String id){
        if (id == null)
        throw new IllegalArgumentException("cannot find
            elements with a null id attribute");

        return new ById(id);

    }
//inner class (static)
public static class ById
{
 public void display()
  {
    =
  }
}
public static void main(String[] args)
 {
  By.ById staticBy = new By.ById();
  staticBy.display();
  }
} //end of By
```

Wrapper classes

* The main objective of Wrapper classes are

→ To wrap primitives into object from so that we can handle primitives also just like objects.

→ To define several utility methods which are required for primitives.

Wrapper classes are :-

① Byte
② Short
③ Character
④ Integer
⑤ Float
⑥ Double
⑦ Long

Constructors :-

Almost all wrapper classes define two constructors. one can take corresponding primitive and the other can take String argument.

ex :- Integer I = new Integer(10); → primitive

Integer I = new Integer("10"); → String

Double d = new Double(10.5); → primitive

Double d = new Double("10.5"); → String

* If the String argument is not representing number then we will get RuntimeException saying NumberFormatException.

ex Integer i = new Integer("five"); → RE: NumberFormatException.

* Float contains 3 constructors with float, double and String.

ex Float f = new Float(10.5f);

Float f = new Float("10.5f");

Float f = new Float("10");

* Character class contains only one constructor with char as argument.

ex Character ch = new Character('r'); ✓

* Boolean class contain 2 constructors with boolean primitive and String argument.

→ If we pass boolean primitive as argument then allowed values are true or false. (where case should be in lowercase).

ex Boolean b = new Boolean(true); ✓

Boolean b = new Boolean(True); ✗

⇒ If we pass String argument then content and case both are not important.

→ If the content is case insensitive String of "true" then it is treated as true. In all other cases it is treated as false.

Ex: Boolean b = new Boolean("true"); → true

Boolean b = new Boolean("false"); → false

Boolean b = new Boolean("True"); → true

Boolean b = new Boolean("Yes"); → false

Note:—

(1) In all wrapper classes, toString() is overridden for to return its content.

(2) In all wrapper classes equals() is overridden for content comparision.

Methods :—

The below are the Utility methods of all wrapper classes.

i) valueOf()

ii) xxxValue()

iii) parseXxx()

iv) toString()

valueOf() :-

→ we can use valueOf() to create wrapper object for the given String or primitive as alternative to constructor:

Form ① :-

→ Every wrapper class except Character class contains the following valueOf() to create wrapper object for the given String

Syntax:- public static wrapper valueOf(String s);

Ex Integer i = Integer.valueOf("10");

Form ②

→ Every Integral type wrapper class (Byte, Short, Integer, Long) contains the following valueOf() to create wrapper object for the given specified radix String.

Syntax:- public static wrapper valueOf(String s, int radix);

→ The allowed range of radix is 2 to 36.

Ex Integer i = Integer.valueOf("111", 2);

Syso(i); // 7

Form ③ :- Every wrapper class including character class defines the following valueOf() to create wrapper Object for the given primitive.

Ex Character ch = Character.valueOf('a');

Boolean b = Boolean.valueOf(true);

xxxValue() :-

→ we can use xxxValue() to find primitive values for the given wrapper object.

→ Every number type wrapper class (Byte, Short, Integer, Long, Float, Double) contains the following xxxValue() to find primitive for the given wrapper object.

```
public byte byteValue()
public int intValue()
public short shortValue()
public long longValue()
public float floatValue()
public double doubleValue()
```

Ex
```
Integer i = new Integer(130);
Syso(i.byteValue()); // -126
Syso(i.intValue()); // 130
Syso(i.longValue()); // 130
```

charValue() :- Character class contains charValue() to find char primitive for the given Character object.

Syntax : `public char charValue();`

Ex
```
Character ch = new Character('a');
char c = ch.charValue();
Syso(c); // 'a'
```

booleanValue();—

Boolean contains booleanValue() to find boolean primitive for the given Boolean object.

syntax: public boolean booleanValue();

ex Boolean b = Boolean.valueOf("defd");
boolean bb = b.booleanValue();
Syso(bb); // false.

parseXxx();—

we can use parseXxx() to convert String to primitive.

syntax①: public static primitive parseXxx(String s);

syntax②: public static primitive parseXxx(String s, int radix);
↓
range 2 to 36

ex int i1 = Integer.parseInt("10");
Double d = Double.parseDouble("10.5");
int i = Integer.parseInt("100", 2);
Syso(i); // 4

toString() :-

we can use toString() to convert wrapper object and primitives to String.

* Every wrapper class contains the following toString() to convert wrapper object to string.

syntax: public String toString();

→ It is the overriding version object class toString().

→ whenever we are trying to print any wrapper object reference internally this method will be called.

ex Integer i = new Integer(20);

Syso(i); // (i.toString()); → o/p : 20

* Every wrapper class including character class contains the following static toString(-) to convert primitive to String.

syntax: public static String toString(Primitive p);

ex String s = Integer.toString(10);

JAVA 1.8 FEATURES

WITH

SELENIUM

Types of Interfaces

① Marker Interface :-

If the interface doesnt have any methods such kind of interfaces are called Marker Interfaces.

ex:- RandomAccess (I)

Clonable (I)

Serializable (I)

② Functional Interface :-

If the interface is having only i.e exactly one abstract method and with any number of default and static method.

ex :

Function (I)

Consumer (I)

Predicate (I)

③ Fully abstract interface :-

If the interface is having purely abstract methods without default and static methods such of kind of interfaces are called as fully abstract interfaces.

ex:

```
interface i1
{
 void m1();
 void m2();
}
```

* In Selenium

WebDriver (I), SearchContext (I)

WebElement (I)

④ Partially abstract interface:

It may or may not contains abstract methods with default and static methods.

Default Methods in Java 1.8 version

In Java 1.8, java introduced inside interface we can have implementations in the form of default methods or static methods. whenever if you want to avoid dependence between implementation classes and interfaces then go for default methods. with concrete implementations.

* How to define default method:

Syntax: default void/returntype methodName ( )
{
=
}

* default is a keyword to convey it is the default method.
* default methods applicable for interface level but not for classes. and if we try to declare at class level immediately we will get compile time error.

ex
```
class A {
    default void m1() → compile time error
    {                   X
    =
    }
}
```

```
interface I1
{
default void m1() ✓
                  ↘ valid
  {
  =
  }
}
```

* whatever we declare default methods as the part of interface by default will be available to every child. and if the child is not satisfied with default implementations then child classes can override that methods.

ex:
```
interface test
 {
default void m1()
  {
  Syso("Hello test");
  }
 }
class TestImpl implements test
 {
  @override
  public void m1()
   {
   Syso("Hello welcome to Rameshsoft"); ✓
   }
 }
```

Note: while overriding default method in child class it should be 'public' but not 'default', otherwise we will get compiletime error.

ex
```
class TestImpl implements test {
    @override
    default void m1() → CE
        {
        =
        }
}
```

* we cannot override Object(c) methods inside interface eventhough concrete implementations are allowed. inside the interface. where as in concrete class or abstract class or in Service Provider class we can override Object(c) methods.

ex
```
interface i1
{
default void m2()
 {
 =
 }
public int hashCode() → CE
  {
   return 1;
  }
}
```

```
class Demo
{
 @override
public int hashCode() → valid
 {
  return 1;
 }
}
```

* If two interface are having common methods with same name and when you provide implementation for that two interfaces we will get ambiguity problems.

* To avoid that ambiguity problems override that method in the child class.

Problem :

```
interface A
{
default void m1()
   {
   =
   }
}
```

```
interface B
 {
 default void m1()
    {
    =
    }
 }
```

```
class C implements A,B
{
public static void main(String[] k)
 {
 C c = new C();
   c.m1(); → CE (ambiguity problem)
 }
}
```

In the above program to avoid ambiguity problem override the common method in the child class.

ex

```
package com.rameshsoft.java18;

interface A
{
  default void display()
      {
        System.out.println("A");
      }
}
interface B
{
 default void display()
    {
    System.out.println("B");
    }
}
public class C implements A,B
{
  public void display()
     {
       System.out.println("Welcome to RameshSoft");
     }
   public static void main(String[] args)
    {
       C c=new C();
        c.display();
    }
}
```

* If you want to access default implementations in child class refer interface name with super keyword.

example:

```
interface A
{
 default void display()
   {
    Syso("A");
   }
}
interface B
 {
  default void display()
    {
     Syso("B");
    }
 }
public class CommonTest implements A,B
 {
   public void display(); ⟶ By using this 'super' keyword
    {                        we can avoid ambiguity.
     A.super.display();
     B.super.display();
    }
  public static void main(String[] k)
   {
   CommonTest c= new CommonTest();
     c.display();
   }
 }
```

static methods in Java1.8

From Java 1.8 version onwards we can define static methods inside interface.

* whenever if it is no where related to object, keep such kind of things or provide implementations as the part of static method.
* we cannot override static methods in child class. as static methods are not available to child class.

Example:-

```
package com.rameshsoft.java18;
interface A
 {
   public static String greetings()
   { return "Rameshsoft"; }
 }
public class CommonTest implements A
 {
   public static void main(String[] args)
       {
       CommonTest c=new CommonTest();
          c.greetings(); ---> CE ------①
          A.greetings(); ------------②
       }
 }
```

Conclusions :

In the above program at line -① we will get compiletime error because interface static methods we cannot access with object reference.

* In the above program at line -② we won't get any compile time error because we can access interface static methods only by using interface names.

* If two interfaces are having the same method names then we won't get any ambiguity problem because static implementations are not available to child class.

Example :

```
package com.rameshsoft.java1.8;
interface A
{public static String greetings()
   {
    return "RameshSoft";
   }
}
interface B
{ public static String greetings()
    {
     return "RameshSoft";
    }
}
```

```
public class CommonTest implements A, B
{
  //not overriding
   public static String greetings()
   {
    return "RameshSoft";
   }
  public static void main(String[] args)
  {
   A.greetings();
   B.greetings();
   CommonTest c=new CommonTest();
    c.greetings();
   }
}
```

How to use default and static methods in Selenium

Example:

```
package com.rameshsoft.java18;
public interface CommonUtility<T extends WebDriver>
{
    public abstract<T> WebDriver getDriver();
```

P.T.O

```
public static void screenshot(WebDriver driver, String path)
  {
   TakesScreenShot t = (TakesScreenshot) driver;
   File file = t.getScreenshotAs(OutputType.FILE);
   FileUtils.copyFile(file, new File(path));
  }
default boolean isVisible(WebElement element)
 {
    boolean b = true;
     try {
        element.isDisplayed();
       }catch(Exception e)
        {
          b = false;
        }
       return b;
 } // end of isVisible(-)
} // end of class
```

Lambda Expressions

Lambda Expressions are introduced in Java 1.8 version.

* What is Lambda expression?

Lambda Expression is an nameless or anonymous function. and which doesn't contain name, return type and access modifiers.

Lambda expressions are also called as anonymous functions or closerres

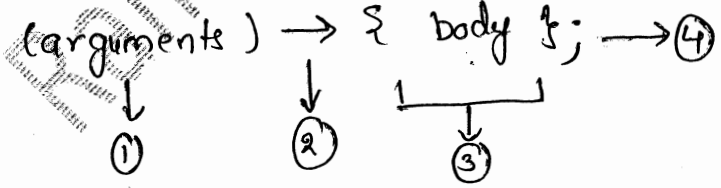
* What is the use of Lambda expression?

→ Less coding i.e, we can reduce length of the code.

→ we can provide implementations for functional interfaces.

* How to define Lambda expressions?

Syntax:



① represents method arguments. and this number of arguments and functional interface abstract methods arguments must be same.

② It is used to link method arguments with body. and to convey it is the Lambda expression.

③ executable statements

④ to convey Lambda expression is ended.

* How to invoke or execute lambda expressions:

we can invoke lambda expressions by using Functional interface.

syntax:

can be anything

interfacename vname = lambda expression;

↓

Should be FunctionalInterface

↓

invoking lambda expression based on FunctionalInterface.

Note:-

→ paranthesis (( )) is optional, if we have only one argument and if we have more than one arguments then ( ) is mandatory.

→ curly braces ({ }) are optional if we have only one statement and if we have morethan one statement then curly braces are mandatory.

Q: Write a Lambda expression to perform click operation?

```
@FunctionalInterface
  interface check
  {
   void click (WebElement e);
  }
```

```
public class Test
{
  public static void main(String[] args)
   {
     check c = (WebElement e)->{ if(e.isDisplayed()&& e.isEnabled())
                                  {
                                    e.click(); } else
                                    { Syso("element is not displayed");
                                  };
     c.click(driver.findElement(By.id("Rameshsoft")));
   }
}
```

Write a lambda expression to perform datatyping actions.

```
@FunctionalInterface
interface data{
   void enterData(WebElement, String data);
   }
public class Test{
   public static void main(String[] args){
 data d = (WebElement e, String t)->{
              if(e.isDisplayed() && e.isEnabled())
               {
                  e.clear();
                  e.sendkeys(t); } else
                 { Syso("element is not displayed"); }
              };
     d.enterData(driver.findElement(By.id("A")), "Rameshsoft");
   } }
```

* Write a Lambda expression to find minimum number?

```
@FunctionalInterface
interface Test
{
  int min(List<Integer> list);
}
public class TestDemo {
  public static void main(String[] args)
  {
    Set<Integer> s= new TreeSet();
    s.add(1000);
    s.add(10);
    s.add(100);
  List<Integer> list = new ArrayList(s);
  Test t = (List<Integer> set) -> {
      return list.get(0);};
    Syso(t.min(list));
  }
}
```

Functional Interfaces

It is new feature came in Java 1.8 version.

*if interface contains only one abstract method such kind of interface is called as Functional Interface. And it is also called as SAM (Single Abstract Method).

Ex① Runnable (I)

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

② Comparable (I)

```
@FunctionalInterface
public interface Comparable<T>
{
    int compareTo(T arg0);
}
```

③ Comparator (I)

```
@FunctionalInterface
public interface Comparator<T>
{
    int compare(T arg0, T arg1);
}
```

④ Predicate (I)

```
@FunctionalInterface
public interface Predicate<T>
{
  boolean test(T arg0);
}
```

⑤ R Function<I>

```
@FunctionalInterface
public interface Function<T,R>
{
    R apply(T arg0);
    =
}
```

* Where we can use Functional interfaces?

→ we can use functional interfaces to invoke lambda expressions, method references and constructor references as well.

* In Functional interfaces in addition abstract method we can have any no. of default methods and static methods.

ex:

```
@FunctionalInterface
interface Test
{
    boolean test(String s);
    default void course(){
    Syso("Java with Selenium in Realtime");
    }
    public static void display()
    {
    Syso("welcome to RameshSoft");
    }
}
```

* In Java 1.8 version to specify and to restrict to Functional interface, they introduced one annotation called @FunctionalInterface

ex: ① @FunctionalInterface
interface Test
{
void check();
}
✓

② @FunctionalInterface
interface SearchContext
{
WebElement findElement(-);
List<WebElement> findElements(-);
}
→ Compile time error

③ @FunctionalInterface
interface Test
{
void check();
default double getCashWithDraw()
{
==
}
public static String miniStatement()
{
==
}
}
✓

Inheritance V/s Functional interface

* Inheritance concept is applicable Functional interface.

ex① @FunctionalInterface
```
interface TestOne
{
  String display();
}@FunctionalInterface
interface TestTwo extends TestOne
{
  String hello();
}
```
→ we will get CE.

① In the above TestTwo having two abstract methods through inheritance. So we will get Compiletime error.

② In the above program if we remove @FunctionalInterface annotation for TestTwo interface we won't get any compile time error.

ex ② @FunctionalInterface
```
interface TestInterface
{

}
```
→ we will get CE bcoz Functional interface should have one abstract method.

How to invoke Lambda expressions using Functional interface?

we can use Functional interfaces to invoke or to execute lambda expressions.

Let us consider one sample example how we can execute lambda expressions using Functional interfaces.

steps to follow :-

① write Functional interface
② write lambda expression
③ invoke lambda expression using Functional interface.

ex①: with using Lambda expression

```
interface Test
{
 void check (int a, int b);
}
public class TestDemo
{
 public static void main(String[] args)
  {
  Test t1=(a,b)->{if (a>b) Syso("a is bigger :"+a);
                  else Syso("b is bigger :"+b); };

   t1.check(89, 86);
  }
}
```

ex ② Without using Lambda expression.

```
@FunctionalInterface
interface Test
{
  void check(int a, int b);
}
class TestImpl implements Test
{
  public void check(int a, int b)
  {
    if(a>b)
    { Syso("a is bigger "+a);
    }else
    { Syso("b is bigger :"+b);
    }
  }
}
public class TestDemo
{
public static void main(String[] args)
  {
    Test t = new TestImpl();
    t.check(89, 86);
  }
}
```

Write a Lambda expression to find length of the String ?

```
package com.rameshsoft.functionInterfaces;

@FunctionalInterface
interface Test                          // without using Lambda expression
{
  int length(String s);
}
class TestImpl implement Test {
  @override
   public int length(String s){
       return s.length();
    }
  }
public class TestDemo {
 public static void main(String[] args)
  {
     Test t = new TestImpl();
     Syso(t.length("Rameshsoft"));
     Syso(t.length("Java with Selenium in Real time"));
     Syso(t.length("9177791456"));
  }
}
```

```
package com.rameshsoft.functional interfaces;

@FunctionalInterface
 interface Test
 { int length(String s);
 }
public class TestDemo {

 public static void main(String[] args){ {
   Test t = (s) -> { return s.length (); };
   Sysa(" t. length ("RameshSoft")];
   Sysa(t.length (" Java with selenium in Real time "));
   Sysa(t.length ("9177791456");
    }
}
```

→ with Lambda expression.

Q:- write a program to find max number using Lambda.

① without using Lambda expression:-

```
@FunctionalInterface
interface Test
{
   int max (List<Integer> list);
}
class TestImpl implements Test{
   @override
    public int max(List<Integer> list)
    { return list.get(list.size()-1); }
   }
```

```
public class TestDemo {
   public static void main(String[] args)
    {
     TestImpl t = new TestImpl();
      Set<Integer> set = new TreeSet();
        set.add(19);
        set.add(5);
        set.add(198);
        set.add(1900);
       List<Integer> list = new ArrayList(set);
       t.max(list);
       }
  }
```

(2) with using Lambda expression:

```
package com.rameshsoft.functionInterface;
@FunctionalInterface
interface Test
 {
  int max(List<Integer> list);
 }
public class TestDemo{
 public static void main(String[] args)
 {
   Set<Integer> s= new TreeSet();
      s.add(10);
```

```
        s.add(1000);
        s.add(1);
        s.add(123);
        List<Integer> list = new ArrayList(s);
        Test t = (List<Integer> set) -> {
        return list.get(list.size()-1);};
        System.out.println(t.max(list));
        }
}
```

Implementing multithreading without Lambda expression.

```
class MultithreadOne implements Runnable {
    public void run(){
        Syso("child thread is started");
        for(int i=0; i<10; i++){
            Thread.yield();
            Syso("child thread"); }}}
public class MultithreadDemo
{
 public static void main(String[] args) throws IE Exception
   {
    MultithreadOne m = new MultithreadOne();
    Thread t = new Thread(m);
    t.start();
    Syso("main thread is started");
    for(int i=0; i<10; i++){ Syso("main thread"); }
    }
}
```

Implementing multithreading with Lambda expression

```
public class MultithreadLambda{
    public static void main(String[] args){
        Runnable runnable = ()-> { syso (" child thread is started");
        for(int i=0; i<10; i++){
            Syso ("child thread");
        }};
        runnable.run();
      Syso ("main thread is started");
      for (int i=0; i<10; i++){
          Syso("main thread");
      }
}
```

Write a Lambda expression to check webelement is displayed or not?

```
@FunctionalInterface
interface TestWebElement
{
 boolean check(WebElement element);

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

```
Syso("Starting main program");
System.getProperty("webdriver.gecko.driver", D://RameshSoft/geckodriver.exe");
WebDriver d = new FirefoxDriver();
d.get("rameshselenium.blogspot.in");

WebElement ele = d.findElement(By.linkText("Tooltip");
WebElement element = d.findElement(By.cssSelector("a[id='u']+
                     input:nth-of-type(2)");

//writing and invoking lambda expression to check element
//is displayed or not
TestWebElement test = (WebElement e) -> { boolean b = true;
                      if(e.isDisplayed() && e.isEnabled())
                      { b = true; }
                      else
                      { b = false; }
                      return b; };
test.check(element);
}
}
```

Predefined Functional Interfaces

There are many predefined functional interfaces are there are the part of

1. BiConsumer(I)
2. Function(I)
3. Predicate(I)
4. Consumer(I)
5. UnaryOperator(I)
6. BinaryOperator(I)
7. Supplier(I)
8. BooleanSupplier(I)
9. IntSupplier(I)
10. IntPredicate(I) ..etc

BiConsumer(I) :-

BiConsumer(I) doesn't return any values and it just accepts inputs.

ex:-
```
@FunctionalInterface
public interface BiConsumer<T, U>{
    void accept(T arg0, U arg1);
    =
}
```

ex public class BiConsumerTest {

public static void main(String[] args)

{

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

m.put(1, "rameshsoft");

m.put(2, "Java");

m.put(3, "selenium");

m.put(4, "real time");

BiConsumer<Integer, String> b = (k, v) -> System.out.println("K"+k+"v"+v);

};

m.forEach(b);

m.forEach((k, v) -> { System.out.println("K"+k+"v"+v); });

}

}

Predicate (I) :-

It's always returns some value based on condition i.e, it always returns boolean value.

ex @FunctionalInterface

public interface Predicate<T>

{

boolean test(T arg0);

=

}

* Whenever we want to execute the program based on condition check then go for Predicate.

ex:

```
public class PredicateDemo {
    static Predicate<WebElement> p1 = t ->
        t.isDisplayed() && t.isEnabled();
    public static boolean isCheck(WebElement e){
        boolean b = p1.test(e);
        return b;
    }
public static void main(String[] x)
{
    WebDriver d = new FirefoxDriver();
    d.manage().window().maximize();
    d.get("https://www.gmail.com");
WebElement element = d.findElement(By.id("IdentifierId"));
 Sysa(PredicateDemo.isCheck(element));
 }
}
```

Function (I):-

Function is an Functional Interface whenever you want to have same response irrespective of type then go for Function.

* Function interface always takes two arguments one is type and one more is return type of type.
* This interface contains one method called apply() and this method can return anything as the part of returntype.

Syntax:

```
@FunctionalInterface
public interface Function<T, R>{
    R apply(T arg0);
    =
}
```

(T → type, R → return type of Type)

Ex

```
public class PredicateFunctionExample {
    static Function<WebElement, String> f1 =
        e -> e.getAttribute("id");
    static Function<WebElement, String> f2 =
        e -> e.getTagName();
    static Predicate<WebElement> p = e -> e.isDisplayed();
    public static void main(String[] args)
    {
```

```
WebDriver d = new FirefoxDriver();
d.manage().window().maximize();
d.get("https://www.gmail.com");
WebElement element = d.findElement(By.id("IdentifierId"));
Syso(f1.apply(element));
Syso(f2.apply(element));
Syso(p.test(element));
}
}
```

double colon operator (::) :

By using this operator we can refer method references as well as constructor reference.

Method Reference :-

whenever we have duplicate code and if we want to avoid duplicate code then go for method reference.

* By using method reference we can get code reusability.
* To refer static methods this is the following.

Syntax : To refer static methods.

```
FunctionInterface vname = classname :: methodname;
```

To refer non-static methods

FunctionalInterface vname = objectRefernce :: methodName;

* method refenence is alternate for lambda expression if we want we can replace lambda expression with method reference.

```
ex  @FunctionalInterface
    interface methodRef
    {
      void m1();
    }
    public class methodReferenceTest {
      public void m8(){
          Syso("123");
          Syso("1");
          Syso("1233");
          Syso("Rameshsoft");
      }
      public static void main(String[] k){
        MethodReferenceTest b = new MethodReferenceTest();
          methodRef m = b::m8;
            m.m1();
      }
}
```

Constructor Reference :-

```
interface i1<T>
 {
  T getTest();
 }
 class A{
   public void m1()
     {
     }
  }//A
 class B {
   public void m()
     {
     }
  }//B
public class C
  {
   public static void main(String[] arg)
     {
     i1<A> i = A::new; // i1<A> i = ()-> { A a = new A();
                                          return a };
      A a = i.getTest();
       Syso(a.m1());

     i1 <B>  i = B::new;
       B b = i.getTest();
        Syso(b.m());
      }
  }
```

UnaryOperator (I):-

By UnaryOperator (I) we can perform single operate.

→ internally UnaryOperator (I) extends Function (I).

pseudocode:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T>
{
   =
}
```

→ internally UnaryOperator (I) uses apply() of Function (I)

Ex

```
public class UnaryTest
{
  public static void main(String[] args)
  {
   int a=9;
  UnaryOperator<Integer> u= e->e*e;
  Syso(u.apply(a));
  }
}
```

Java 8 - New Date/Time API

Java 8 new Date/Time API introduced to cover the following drawbacks of old date-time API.

* Not thread-safe :- java.util.Date is not thread safe, thus the developers have to deal with concurrency issue while using Date. The new ~~time~~ API Date-time API is immutable and doesn't have setter methods.

* API's design and understanding :- The Date and Calender API's are loosely designed with inadequate methods to perform day to day operations. The new Date Time API is follows consistent domain models for Date Time Duration and Periods. There are wide variety operations are of utility methods that supports the common operations.

* Timezone handling :- Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

Java 8 introduces a new date-time API under the package java.time.

The classes are available in java.time package are

1. LocalDate (c)
2. LocalDateTime (c)
3. LocalTime (c)
4. ZonedDateTime (c)
5. DateTimeException (c)
6. MonthDay (c)

etc

LocalDate, LocalTime and LocalDateTime

* The most commonly used classes are LocalDate (c), LocalTime (c), and LocalDateTime (c).
* As their names indicate, they represent the local Date/Time from the context of the observer.
* These classes are mainly used when timezone are not required to be explicitly specified in the context.

LocalDate :-

The LocalDate represents a date in ISO format (yyyy-MM-DD) without time. It can be used to store dates like birthdays and pay days.

Example:-

```
public class LocalDateTest{

    public static void main(String[] k){

        LocalDate localDate1 = LocalDate.now();

        Syso(localDate1);

        Clock clock = Clock.systemDefaultZone();
        LocalDate localDate2 = LocalDate.now(clock);
        Syso(localDate2);

        ZoneId zoneId = ZoneId.of("Europe/paris");
        LocalDate localDate3 = LocalDate.now(zoneId);

        Syso(localDate3);

        LocalDate localDate4 = LocalDate.of(2017, 09, 16);

        Syso(localDate4);

        LocalDate localDate5 = LocalDate.parse("2017-09-16");

        Syso(localDate5);

        LocalDate localDate6 = localDate5.plusDays(2);

        syso(localDate6);

        LocalDate localDate7 = localDate5.minus(2, ChronoUnit.DAYS);

        Syso(localDate7);
    }
}
```

LocalTime :-

The LocalTime (c) represents time without a date,

Similar to LocalDate an instance of LocalTime can be created from system clock or by using parse(-) and of(-).

→ An instance of current LocalTime can be created from the System clock as below.

```
LocalTime now = LocalTime.now();
```

→ In the below code sample, we can create a LocalTime representing 06:30 Am by parsing a string representation.

```
LocalTime sixthirty = LocalTime.parse("06:30");
```

→ The Factory method "of()" can be used to create a LocalTime.

```
LocalTime sixthirty = LocalTime.of(6,30);
```

example:-

```
public class LocalTimeTest {
    public static void main(String[] args) {
        LocalTime currentTime1 = LocalTime.now();
        Syso(currentTime1);
        LocalTime currentTime2 = LocalTime.parse("09:30");
        Syso(currentTime2);
```

```
LocalTime of = LocalTime.of(9, 45);
System.out.println(of);
LocalTime plus = of.plus(1, ChronoUnit.HOURS);
Syso(plus);
boolean before = LocalTime.parse("09:30").isBefore(LocalTi
                                    -me.parse("09:35"));
System.out.println(before);
System.out.println(LocalTime.MAX);
System.out.println(LocalTime.MIN);
}
}
```

LocalDateTime (c);

The LocalDateTime is used to represent a combination of date and Time.

→ This is the most commonly, class used when we need a combination of date and time. The class offers a variety of APIs and we will look at some of the most commonly used ones.

→ An instance of LocalDateTime can be obtained from the System clock similar to LocalDate and LocalTime.

LocalDateTime.now();

→ The below code explains how to create an instance using the factory "of()" and 'parse()' methods.

LocalDateTime.of(2017, Month.SEPTEMBER, 16, 05, 30);

LocalDateTime.parse("2017-09-16 T05:30:00");

Example:-

```
public class LocalDateTimeTest{
    public static void main(String[] args){
        LocalDateTime localDateTime = LocalDateTime.now();
        System.out.println(localDateTime);
        LocalDateTime localDateTime2 = LocalDateTime.of(LocalDate.now(),
                                        LocalTime.now());
        System.out.println(localDateTime2);
        LocalDateTime localDateTime3 = LocalDateTime.parse("2017-09-16
                                        T10:45:50");
        System.out.println(localDateTime3);
        LocalDate localDate = localDateTime3.toLocalDate();
        LocalTime localTime = localDateTime3.toLocalTime();
        System.out.println(localDate);
        System.out.println(localTime); }
}
```

ZonedDateTime :-

Java 1.8 version provides ZonedDateTime when we need to deal with zone specific date and time.

* The ZoneId is an identifier used to represent different zones.
* There are different time zones and the ZoneId are used to represent them.

Example :-

```
public class ZonedDateTimeTest{
    public static void main(String[] args){
      ZoneId zoneId = ZoneId.of("Asia/Tokyo");
    ZonedDateTime zoneDateTime = ZonedDateTime.of(LocalDateTime.
                                        now(), zoneId);
    Syso(zoneDateTime);
  LocalDateTime localDateTime = LocalDateTime.of(2017,
                          Month.SEPTEMBER, 20, 07, 40);
    ZoneOffset offset = ZoneOffset.of("+02:00");
    OffsetDateTime offsetDateTime = OffsetDateTime.of(localDateTime,
                                        offset);
    Syso(offsetDateTime); }
}
```

Stream API

Stream:- It is a sequence of objects from a source which supports aggregation operations.

* A Stream takes collection, arrays, IO resources as input.
* A stream supports aggregate operations like filtering, mapping limit, find, match ... so on.
* Stream is present in java.util.stream package.
* java.util.stream is a package which contains classes, interfaces for processing sequence of elements.

Ways to get Stream object:-

Way1:- we can get the stream from Arrays().

Syntax:

```
class Arrays
{
    public static <T> Stream<T> stream(T[] args)
    {
    =
    }
}
```

ex

```
String[] s = {"Java", "Rameshsoft", "selenium"};
Stream<String> stream = Arrays.stream(s);
```

↳ it is an utility class from java.util package.

Way2:- By using of() of Stream(I).

* Stream is having two overloaded of() methods.

Syntax:-
```
interface stream
{
  static default<T> Stream<T> of(T arg)
    {
    =
    }
  public static<T> Stream<T> of(T... arg)
    {
    =
    }
}
```

ex:- Stream stream = Stream.of("Java", "Rameshsoft", "selenium");

Way3:- As part of java 1.8 Collection(I) is having a default method called stream().

Syntax:-
```
interface Collection {
    default Stream<E> stream()
      {
      =
      }
}
```

Example:-

```
public class StreamTest {
    public static void main(String[] args)
    {
        //1st way
    List<String> list = new ArrayList();
     list.add("java");
     list.add("selenium");
     list.add("rameshsoft");
     Stream<String> s= list.stream();
      s.forEach(System.out::println);
      //2nd way
    String[] a ={"1","6","9","35","89"};
    Stream<String> arr = Arrays.stream(a);
      //arr.forEach(System.out::println);
      //3rd way (1 parameter of(T t) method
      Stream<String> s1=Stream.of(a);
       s1.forEach(System.out::println);

      //var arg of(T...t) method
      Stream<String> s2 = Stream.of("Java","selenium","Rameshsoft");
       s1.forEach(System.out::println);
    }}
```

Filtering :-

Java Stream provides a method filter() to filter stream elements on the basis of given predicate.

* This method takes predicate as an argument and returns a stream of consisting of resulted elements.

Syntax:

```
Stream<T> filter (Predicate<? super T> arg);

  public interface Stream<T>
    {
      Stream<T> filter (Predicate<? super T> arg);
        =
    }
```

ex: public class StreamFilterTest {

```
    public static void main(String[] args)
      {
        List<Course> course = new ArrayList();
        course.add (new Course ("Java", 1000, "ramesh", 2));
        course.add (new Course ("selenium", 2000, "ramesh", 2));
        course.add(new Course ("appium", 2000, "ramesh", 2));
        course.add(new Course("webservices", 1000, "harish", 2));
```

```
course.add(new Course("spring", 1000, "ramesh", 2));
course.stream().filter(p->p.getFee()>1000).forEach(System.out::
                                                    println);

boolean b= course.stream().allMatch(p->p.getCourseName().contains
                                                    ("java"));
System.out.println(b);

boolean c= courses.stream().anyMatch(p->p.getCourseName().
                                    contains("java"));
System.out.println(c);

boolean d= course.stream().noneMatch(p->p.getCourseName().
                    contains("java"));
System.out.println(d);
}
}
```

Optional :-

Java 1.8 has introduce a new class called optional in Java.util package.

* It is used to represent a value is is present or absent.

* The advantage of this class is, it avoids any runtime NPE. (Null_PointerException)

* It is also a container to hold at most one value.

Advantages:-

1. we can avoid NPE at runtime.
2. Null checks are not required.
3. we can hold up one value.

pseudocode:-

```
public final class Optional<T>
  {
    public <U> Optional<U> map(Function<? super T, ?
                    extends U> argo)
        {
        =
        }
    public<U> Optional<U> flatMap(Function<? super T, Optional<U>>
                                    argo)
        {
        =
        }

    public static <T> Optional<T> empty() {
                =
        }
```

```
public static <T> Optional<T> of(T arg)
{
=
}
public static <T> Optional<T> ofNullable(T arg)
{
=
}
}
```

Example1:-

```
public class OptionalDemo{
public static void main(String[] args)
{
Optional<String> gender_optional= Optional.of("MALE");
String answerA ="Yes";
String answerB= null;
System.out.println("Non-empty optional:" + gender_optional);
Syso("Non-empty optional : Gender value : "+ gender_optional.get());
Syso ("Empty optional :"+ Optional.empty());
Syso ("ofNullable on Non-empty optional :"+ Optional.ofNulla-
                                        -ble(answerA));
Syso("ofNullable on Empty optional : " + Optional.ofNullable(answerB));
Syso ("ofNullable on Non-empty optional :" + Optional.of(answerB));
}}
```

Example2:—

```
public class OptionalMapDemo {
    public static void main (String[] args)
    {
    Optional<String> nonEmptyGender = Optional.of("male");
    Optional<String> emptyGender = Optional.empty();
Syso("Non-empty optional: " + nonEmptyGender.map(String::
                                    toUpperCase));
Syso("empty optional: "+ emptyGender.map(String::toUpperCase));
Optional<Optional<String>> nonEmptyOptionalGender = Optional.of(
                                    Optional.of("male"));
Syso("optional val:" + nonEmptyOptionalGender);
Syso("optional map :" + nonEmptyGender.map(gender ->
                    gender.map(String::toUpperCase))
Syso("optional flatmap :" + nonEmptyOptionalGender.flatMap(
                    gender -> gender.map(String::toUpperCase)));
    }
}
```